Behavioural and Societal Sciences
Kampweg 5
3769 DE Soesterberg
P.O. Box 23
3769 ZG Soesterberg
The Netherlands

www.tno.nl

T +31 88 866 15 00
F +31 34 635 39 77
infodesk@tno.nl

TNO report

TNO-DV 2011 IN433

CHAOS 2.0 Behaviour and Performance Modelling Framework

Date	March 2012
Author(s)	drs. E.M. Ubink drs. R. Looije
Number of pages	28 (incl. appendices)
Sponsor	PGL
Project name	CHAOS 2.0 Behaviour and Performance Modelling Framework
Project number	053.01287

Contents

1 Background

CHAOS (Capability-based Human-performance Architecture for Operational Simulation, (Ubink, Aldershoff, Lotens, & Woering, 2008), (Ubink, Lotens, & Woering, 2010) is a Java software library that contains a behaviour modelling architecture. It can be used to create human behaviour and performance models. CHAOS was originally developed for the SCOPE simulation environment, a simulation of dismounted soldier operations (Ubink, Aldershoff, & Lotens, 2008). However, it is a generic framework that can be used for other domains and applications as well.
Currently, it is central part of the simulation environments SCOPE, SCOPE Light, BRIGADE and the Driver Model Library. SCOPE Light is derived from SCOPE and focuses on the simulation of physical performance, thermal strain and clothing systems. BRIGADE is a simulation of fire fighting operations. BRIGADE is still in a prototype stage, but further development of the simulation has been postponed. The Driver Model Library (DML) is a new initiative that is still in full development. The aim of the DML is to provide a collection of models for traffic behaviour (drivers) that can be used in various (third party) traffic and driving simulations.

1.1 CHAOS 2.0

By applying CHAOS in simulation environments over recent years, some problems and shortcomings of the original CHAOS design have emerged. This has resulted in the development of CHAOS 2.0, a new version of CHAOS that offers many improvements in the internal mechanisms, class structure and interface definitions. All these changes result in improved usability and should make developing and maintaining behaviour and performance models much easier.

1.2 About this report

The CHAOS library is implemented in Java, a third generation object oriented programming (OOP) language. For some sections of this report, especially chapters 3 and 4, a basic understanding of OOP is assumed. More information on OOP and Java can be found in the online Java Tutorials (Oracle, 2011).

1.2.1 Naming conventions

To improve readability, all code fragments, including package, class and method names, are written in the `Courier` font. Also, the standard Java naming conventions are used throughout this report:

- an object definition is called a *class,*
- an instantiation of a class is called an object,
- the functions of a class are called methods,
- classes are organized in packages,
- an interface is a reference type that contains no method bodies. Interfaces cannot be instantiated and can only be implemented by classes, or extended by other interfaces,
- an abstract class is a class that cannot be instantiated but can be extended. Unlike an interface, an abstract class can contain implemented methods.

- J*avadoc* is a code documentation tool provided with Java that, generates documentation in HTML format from code comments. The CHAOS library contains documentation generated by Javadoc.

1.2.2 Report structure

Readers who are looking to get a broad overview of the CHAOS framework are advised to just read Chapter 2. It provides a general introduction into the modelling framework. It also describes some of the shortcomings of the original design that have led to the development of the updated version, CHAOS 2.0. The most important changes between the original and new version are also globally described.

Readers with interest in the technical design should continue reading Chapter 3. It describes the new version of the framework in more detail, with object descriptions and class diagrams.

Finally, programmers who want to start working with the CHAOS framework are advised to read all chapters, including Chapter 4. This final chapter provides a “getting started manual”, focused on programmers and modellers.

2 Introduction to CHAOS

CHAOS is a Java software library that can be used in conjunction with simulation environments to provide human behaviour and performance models for the agents in the simulation. The framework provides generic objects that need to be sub-classed to represent the behaviours and resources that are specific to the simulation. Once these subclasses are implemented, CHAOS takes care of the dynamics in the system by continuously deciding which behaviours are active and which are not.

Figure 1 The CHAOS logo.

2.1 Competition of different behaviours as a multi-agent system

The CHAOS behaviour architecture is based on the idea that humans are either driven by goals they want to achieve (proactive behaviours), or by events that require them to react in a certain way (reactive behaviours). Proactive behaviours are related to tasks, while reactive behaviours are triggered by external events such as a phone that rings, an enemy that appears and starts shooting, or by the person's own status, e.g. in case of fatigue or anxiety.
The idea behind CHAOS is that all (latent) proactive and reactive behaviours are in constant competition with each other. These behaviours are implemented as software agents, called *demons.* The goal of these demons is to influence the behaviour of the entity. In order to do this, they need to obtain the *resources* that they require. These resources represent human capabilities, such as reasoning, fine motoric control, or aerobic capacity.

2.2 Priority of behaviour

The competition between the demons is won by the demons that are most important in the current situation. The demons communicate their importance by "shrieking": the louder a demon shrieks, the more important it is. The shrieking level of proactive demons is usually static and reflects the priority of the task they represent. The shrieking of reactive demons is dynamic and depends on external factors. For instance, when an enemy is detected, this is a very important event that requires an appropriate reaction.

The demon that models this reactive behaviour (e.g. taking cover, returning fire) will in this situation shriek very loudly, probably louder than the other demons in the competition, which would make it the winner of the competition. This means that it can take the resources it requires to influence the behaviour of the entity.

2.3 Multi-tasking

Although a single demon will be shrieking loudest because it is most important, this does not mean that the other demons in the competition cannot influence behaviour as well. The only requirement for influencing behaviour is that the *required* resources are available. If two demons are not conflicting with regards to the resources they require, they can execute their behaviours simultaneously.
Note that it is possible that not enough resources are available for *optimal* performance, in which case sub-optimal multi-tasking may occur. Note that this does require that the demons are capable of mapping available resources to performance. In other words, the demons would need to contain a performance model that takes resource levels as input and gives task performance as output.

2.4 Stress

In CHAOS, stress can be represented as a reactive demon that monitors a specific (set of) model variable(s). For instance: to model stress related to the thermal condition of a person, a “heat stress demon” could be implemented. This demon monitors the person’s body temperature. As the temperature crosses a predefined threshold, the demon starts shrieking. As body temperature rises, shrieking level will increase, until the maximum stress level is reached (see Figure 2).
The maximum stress level indicates that a further increase in body temperature will not result in a further reduction of performance. In other words, the demon “knows” in what range body temperature affects performance. The stress level is reflected by the demon’s shrieking level.

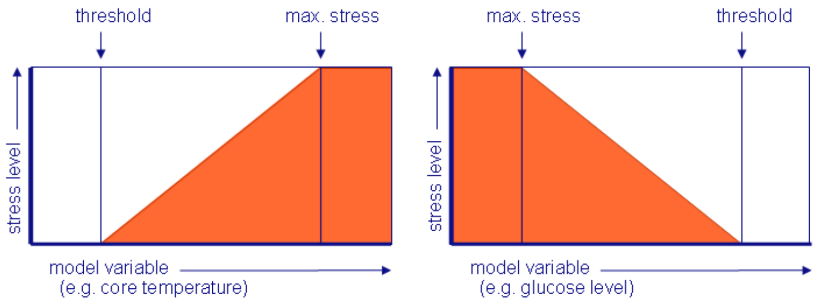

Figure 2 Linear stress modelling in CHAOS. The shrieking level of a stress demon corresponds to the stress level, which is in turn connected to model variables.

The next step is that the demon affects certain resources, depending on its shrieking level. This will in turn affect the performance of the behaviour demons that require these resources, effectively turning stress into strain. Through this mechanism, the effect of stress on performance can be modelled in CHAOS.

2.5 Main algorithm

The central algorithm in CHAOS is essentially a four-step procedure that is repeated each time-step. By sub-classing demons and resources, specific behaviour and performance models can be created. The general mechanism for deciding which behaviours are active will be the same for every application and is described in these four steps:

1. **Reset resources**
   The resources may be affected as a result of the previous iteration, so the resources are reset to their default levels. Also, effects that traits may have on resources can be effectuated in this step.
2. **Shrieking levels are adjusted**
   This step is only relevant for reactive demons with dynamic shrieking levels: they can adjust their shrieking level according to the current state of the world and/or the agent/entity.
3. **Stress-demons affect resources**
   The demons that represent some form of stress can affect (increase or decrease) resource levels, according to their shrieking level, i.e. according to the stress level. This could be viewed as the transformation of stress into strain.
4. **Execute actions**
   In this step, demons are requested to take actions, starting with the demon that is shrieking loudest. This demon will determine if the resources it requires are available. If so, it will take these resources and execute its behaviour. If not, it will do nothing. Then the next demon in line (the loudest demon of the rest), checks if the resources it requires, are available. If so, it will take these resources and execute its behaviour. This process continues until the last demon has had a chance to execute its behaviour.

2.6 Issues with original design

The original idea for CHAOS is simple: create a competition (the pandemonium) between “behaviour elements” (demons). At stake in the competition are “resources”. When a demon succeeds in collecting enough resources, it is allowed to control behaviour.
This still is the central mechanism in CHAOS, also in CHAOS 2.0. However, by applying CHAOS in a number of simulation environments, a number of “specialized” demons have evolved, each with specific functionalities. The problem is that these “specialized” demons are all still based on the original, generic demon template, which was not developed with these different functionalities in mind. This can make it difficult to fathom existing behaviour models and can lead to confusion when developing new demons.
The original CHAOS API (Application Programmers Interface) does not provide much guidance as to what types of “demon specializations” are available or should be used in certain situations. The API also does not provide insight in how certain types of demons should be implemented. The goal of CHAOS 2.0 is to untangle the different specializations and provide specific support for them. In the following sections we shall look briefly at what these specializations consist of and what issues they may cause. In section 2.7 we shall look briefly into the changes made to CHAOS 2.0, intended to solve the issues identified here. Readers that require more technical insight are encouraged to read Chapter 3, in which the CHAOS 2.0 design is described in more detail.

2.6.1 Hierarchical behaviour

In CHAOS, complex behaviours are often modelled as a collection of demons, rather than as a single demon, with each demon in the collection being responsible for a specific element (subtask) of the complex behaviour. These collections of demons are hierarchically structured in parent-child relations.
The parent demons can be viewed as “managers”, while the children may be seen as “worker demons”. Another way to look at it is as contractors and subcontractors. The managers/contractors are responsible for updating the shrieking level and collecting the data (from the simulation) required by the worker/subcontractor demons. They also determine when a certain worker demon should become active. The worker demons contain the actual algorithms for controlling behaviour.
They also know which resources are required.
For example, in the SCOPE simulation of soldier operations, the behaviour involved in reacting to enemy threat is structured hierarchically. There is a manager demon that is concerned with assessing the threat level and that adjusts its shrieking level accordingly. It has two worker demons, a demon that is capable of finding and taking cover and another demon that is capable of shooting at an enemy. So the manager assesses the situation and instructs the workers, and the workers take action by seeking cover and firing at the enemy.
The problem with this original setup of CHAOS is that both worker and manager demons are based on the same, generic demon template, while they have different responsibilities and provide different functionalities. Depending on the specific role a demon has, some of its generic functionality will be implemented while other aspects are unspecified. This can result in confusing code that is difficult to grasp.

2.6.2 States

Some demons in CHAOS make use of states, such as “in progress”, “paused” or “finished”. The behaviour of the demon depends on the state it is in. State transitions are taken care of by either the demon itself or by its manager.
The manager is notified when one of its worker demons changes state, so it can react accordingly. A problem with this approach is that quite some communication between demons takes place that is related to these state changes. Parent demons need to be able to react appropriately to all kinds of state changes, but the API provides no clear instructions on this. Therefore it is the responsibility of the model builder/developer, who has to make sure that all situations are handled correctly. If he/she forgets to handle some relevant state changes, the demons will not behave as expected.

2.6.3 Proactive and reactive demons

A demon’s priority or importance is reflected by its shrieking level. Some demons have a static shrieking level that represents the intrinsic importance of that demon in that specific scenario. They typically represent goal directed behaviour, such as a task that needs to be performed. These demons can also be viewed as proactive, in the sense that they have a goal they want to achieve. As the opposite of a proactive demon, a reactive demon reacts to its environment and therefore has a dynamic shrieking level that is modulated according to changes in the environment. Again, the problem with the original version of CHAOS is that proactive and reactive demons are implemented in the same way. The architecture does not provide separate classes or interfaces to distinguish the two, which can lead to confusion.

2.7 Changes in CHAOS 2.0

To improve the CHAOS architecture and solve the identified issues, a number of changes has been made to the original CHAOS architecture. Besides numerous minor changes, the following three major changes to the class structure of CHAOS have been made (see also Figure 3):

1 Proactive and reactive demons are now supported by specific subclasses of the generic demon class, with distinct functionality.
2 The confusing role of child (or "worker") demons in a parent/child structure has been replaced by dedicated "Behaviour" components. This approach is based on the strategy design pattern (Gamma, Helm, Johnson, & Vlissides, 1995) (Grand, 2002), which means that the Behaviour components form different strategies that can be employed by the demon.
3 To accommodate the need for states, the behaviour objects can optionally be associated with states. The states are implemented according to the state design pattern (Gamma, Helm, Johnson, & Vlissides, 1995) (Grand, 2002), which is definitely an improvement over the previous solution.

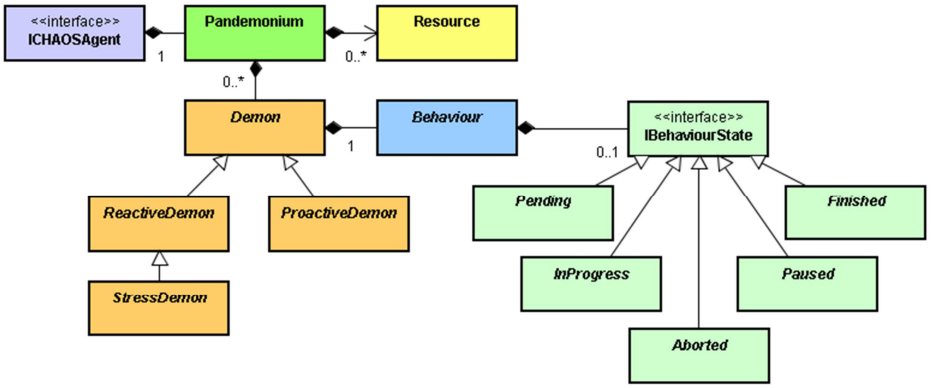

Figure 3 The CHAOS 2.0 class diagram, showing the most important classes. The major changes are the separate classes for *ReactiveDemon* and *ProactiveDemon* and the addition of dedicated *Behaviour* components that can optionally be extended with states.

So the major improvements in CHAOS 2.0 are the untangling of the different roles demons play, by the addition of dedicated classes for these roles. The result is that there are more classes now, but the classes themselves have become less complicated. This makes it much easier to understand what a class is supposed to do and how a class can be extended to provide custom functionality. For a more detailed information on the CHAOS 2.0 classes and their functionality, the reader is referred to Chapter 3.

3 CHAOS 2.0 Design Description

In this chapter, the global design of the CHAOS framework, version 2.0, is described. For the sake of simplicity, only the most important classes and methods are included here. Classes that are not directly related to behaviour modelling, such as GUI components and classes for event passing, are not described here. Some of these classes will be treated in Chapter 4. For a complete listing of classes and methods, the reader is referred to the Javadoc documentation provided with the CHAOS library.
Another thing to clarify beforehand is that CHAOS is a software library that is intended to be used in conjunction with other software, usually simulation software (e.g. a simulation of soldiers or drivers). CHAOS provides behaviour modelling functionality that can be used to control the agents in such a software simulation. When the term “simulation” is used, it therefore refers to external software that is not part of CHAOS, but that uses CHAOS for its behaviour modelling.

3.1 Overview of modelling classes

The classes that are related to the modelling aspects of CHAOS are shown in the UML diagram in Figure 4. In the diagrams in this chapter, the relations with a black diamond denote compositions: a `Pandemonium` *is contained in* an `ICHAOSAgent`, the `Pandemonium` *contains* `Demons` and `Resources` and a `Demon` *contains* a `Behaviour` component, etc. Relations with a white arrowhead at the end denote specialization: a `StressDemon` *is a* `ReactiveDemon`, which *is a* `Demon`, etc. Also, the names of abstract classes (i.e. classes that cannot be instantiated and therefore need to be extended) and abstract methods (methods in an abstract class that need to be implemented by a subclass) are printed in *italic,* but only in the UML diagrams, not in the running text.

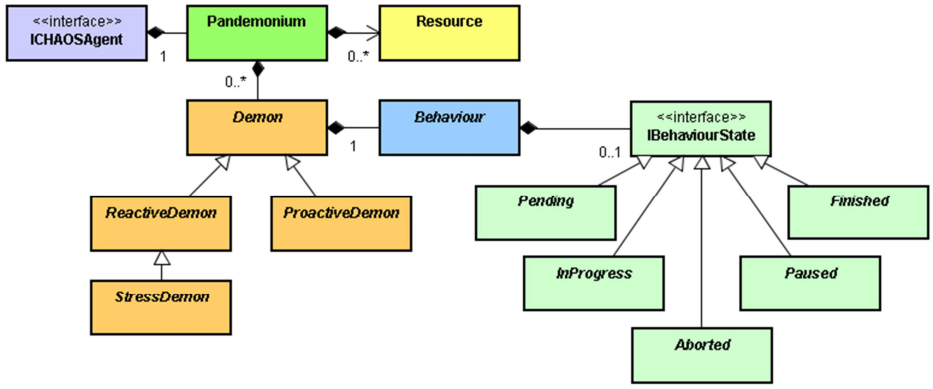

Figure 4 UML class diagram of most important modelling classes in CHAOS 2.0.

In the following sections, all these classes are described separately. However, for the sake of simplicity, not all functionality shall be described. For a full listing and description of all methods, the reader is referred to the Javadoc documentation that is provided with the CHAOS software library.

3.2 ICHAOSAgent

The `ICHAOSAgent` class (Figure 5) plays an important role in CHAOS, as it forms an important connection between the external simulation and the CHAOS framework. It is prefixed with an 'I' to indicate that it is not a class but an interface that still needs to be implemented. This interface should be implemented by an object that has access to an agent in the simulation that the CHAOS framework is used for. For each agent in the simulation that is to be controlled by CHAOS, an `ICHAOSAgent` should exist, each with its own pandemonium and its own demons.

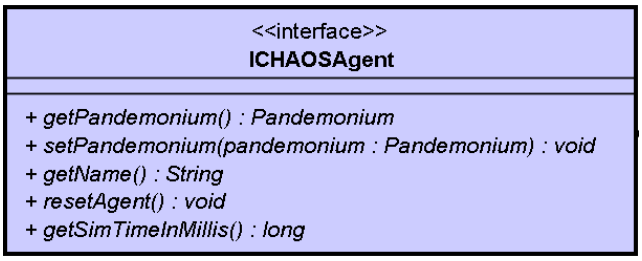

Figure 5 The `ICHAOSAgent` interface.

The most important functionality that `ICHAOSAgent` provides is access to the simulation time from CHAOS, and access to the pandemonium from the simulation. The simulation time is required by several objects in CHAOS, mostly demons, for instance to determine how much time has passed since the last update.
The simulation needs access to the agents pandemonium because the simulation is responsible for calling the `update()` method of `Pandemonium`, that results in execution of the main loop (see also Section 3.3).
Besides this generic functionality, the object that implements `ICHAOSAgent` may be a good candidate to provide the specific functionality that is required for the simulation that CHAOS is connected to. In other words, methods could be added that the demons can call to control the agent's behaviour. Also, methods could be provided that allow the demons to query (aspects of) the state of the simulated world. However, since for each simulation specific demons are developed, it may be easier to allow the demons direct access to the relevant aspects of the simulation. For more information on this subject, the reader is referred to Chapter 4.

3.3 Pandemonium

The `Pandemonium` (Figure 6) class plays a central role in the CHAOS modelling framework. The `Pandemonium` has three main functions:

1 it contains the demons that are competing;
2 it contains the resources the demons are fighting over ;
3 it manages the competition between the demons.

The methods of the `Pandemonium` class are closely related to these three tasks. There are methods to add and remove demons and resources and methods that provide access to the resources, so the demons can inspect which resources are available, and can take the resources they require. This functionality will be described in more detail in Section 3.4. The `Pandemonium` class also provides the possibility to add so called observer objects, `ICHAOSEventObserver` objects to be precise. These observers can register with the pandemonium to receive an

event each time the `Pandemonium` is updated. This functionality is mostly used by GUI components.
Finally, the `Pandemonium` contains an `update()`method that is called from the simulation, usually from the main simulation loop and on each time-step.
This method triggers the execution of an iteration of the main `Pandemonium` algorithm, that is described in Section 2.5.

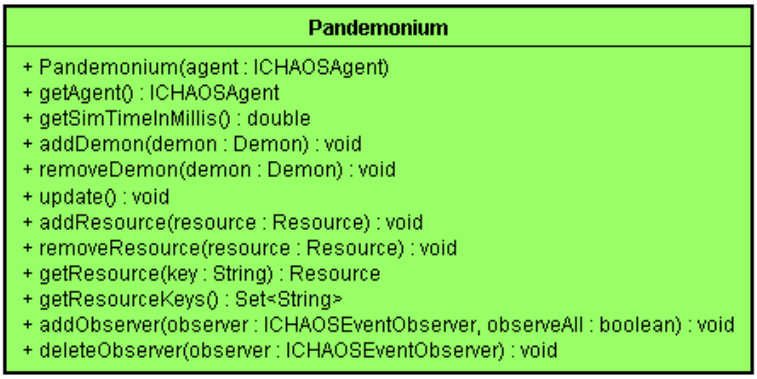

Figure 6 The `Pandemonium` and its most important methods.

3.4 Resource

A `Resource` (Figure 7) in CHAOS is a very simple object. It can be viewed as a container, much like a bucket that contains water. It has a minimum, maximum and an actual level. It has a "key" by which it can be retrieved from the pandemonium, and a name for displaying purposes. The actual "resource" can be taken out of the container in two ways: by calling `limitLevel(int,Demon)` or by calling `lowerLevel(int,Demon)`.

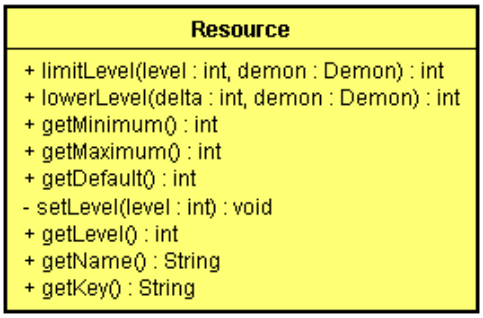

Figure 7 The `Resource` class with its most important methods.

The method `limitLevel(int,Demon)` is called by demons that represent stress. It limits the level of the resource to the level specified by the first integer parameter. If the old level was less than, or equal to the specified level, nothing happens.
The second method (`lowerLevel(int,Demon)`) lowers the level with the delta specified in the first parameter, but never lower than the minimum level of the resource. This method is called by demons that represent behaviour. Both methods return the amount by which the resource was lowered, which is possibly zero.

3.5 Demon

Another key class in CHAOS is the `Demon` class (Figure 8). This class has become significantly less complex in CHAOS 2.0, since much of its original functionality has been defined in subclasses or has been "outsourced" to the `Behaviour` class, that will be described in Section 3.7.

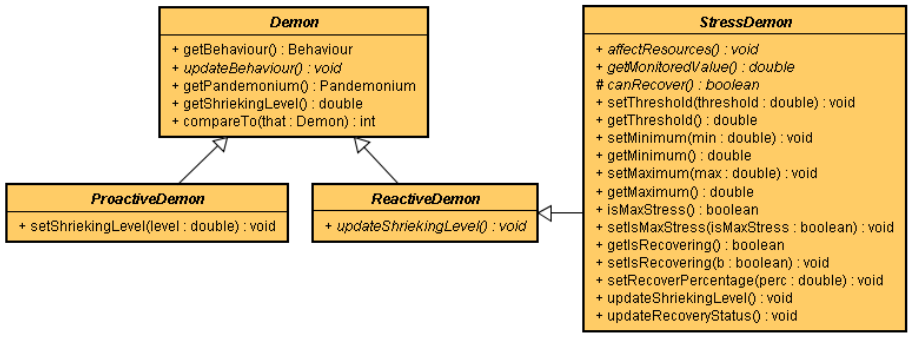

Figure 8 The `Demon` class with subclasses.

`Demon` is an abstract class, which means it cannot be instantiated directly. It has an abstract method that needs to be implemented by subclasses: `updateBehaviour()`. This method is called from the main pandemonium algorithm (see section 2.5), prior to the step in which the resources are taken. When this method is called, the `Demon` can decide if its "strategy", i.e. its `Behaviour` component, should be replaced with another `Behaviour` component, or if some of the current `Behaviour`'s parameters need to update. The method `compareTo(Demon)` compares the demon with the demon that is passed as a parameter. It returns -1, 0 or 1, depending on which demon is shrieking louder.
The other methods are self-explanatory.

3.5.1 *ProactiveDemon and ReactiveDemon*

The abstract subclasses `ProactiveDemon` and `ReactiveDemon` only differ with respect to how the shrieking level is managed. The shrieking level of a `ProactiveDemon` is set externally, by a call to `setShriekingLevel(double)`. A `ReactiveDemon` should manage its shrieking level autonomously, by implementing `updateShriekingLevel()`.

3.5.2 *StressDemon*

The most complex class in Figure 8 is `StressDemon`. This abstract subclass of `ReactiveDemon` can be used to represent stress. In CHAOS, a `StressDemon` monitors a stressful situation and adjusts its shrieking level according to the seriousness of the situation. The more stressful the situation is, the louder the `StressDemon` will shriek. See Figure 9 for an illustration of this process, that is also explained in more detail in Section 2.4.
To support this mechanism, `StressDemon` provides some generic functionality. For this functionality to work, subclasses should be able to describe the seriousness of the situation in a single number (`getMonitoredValue()`), along with a minimum, maximum and threshold value. When these data are supplied, the `StressDemon` will autonomously calculate its current shrieking level.
The stressor can be reversed (e.g. the right side of Figure 9) by calling `setIsMaxStress(false)`.

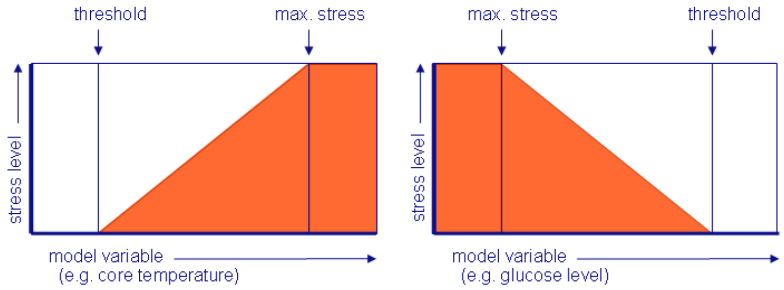

Figure 9 Linear stress modelling in CHAOS. The shrieking level of a stress demon corresponds to the stress level, which is in turn connected to model variables.

Another aspect that is related to stress is recovery, for which the `StressDemon` class also provides functionality. The idea behind the recovery process is that as long as the stress level rises, the `StressDemon` will continue to increase its shrieking level. At a certain point it will be allowed to take some resources. These resources may then be used for behaviour that aids in the recovery from the stressful situation. In other words, it can use its `Behaviour` component to recover from the stressful situation.

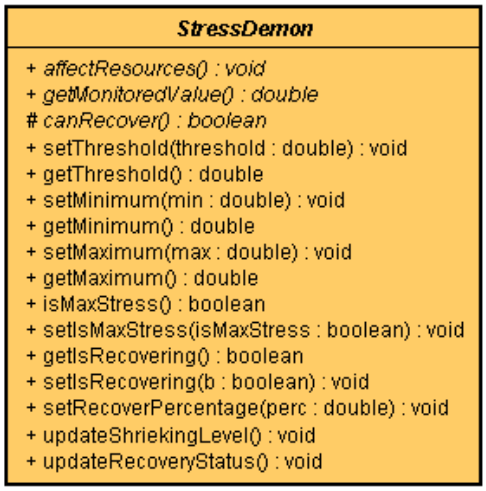

Figure 10 The `StressDemon` class.

For example, if there is a stressor that models physical strain, such as fatigue, the associated recovery behaviour may be “resting”, i.e. preventing that physical activity takes place, which would further increase fatigue. However, a problem that may occur with this recovery solution is that, as soon as recovery starts, the seriousness of the stressful situation decreases, which in turn decreases the shrieking level of the `StressDemon`. This could then result in the resumption of the normal behaviour, which would in turn increase the stress level (e.g. fatigue), which would then trigger the recovery behaviour, etc. In other words, the behaviour of the agent would start to oscillate rapidly, between the normal and recovery behaviours.
To prevent this type of rapid oscillation, the `StressDemon` class provides the method `setRecoverPercentage()`.
The effect of calling `setRecoverPercentage()` is that the `StressDemon` stops recovering only after the stressfulness of the situation has decreased with the

specified percentage. For instance, only after fatigue has decreased with 10% will the `StressDemon` lower its shrieking level, which would then result in the normal behaviour taking over again. Although this still results in oscillations, it does allow to dramatically limit the oscillation frequency.
Besides calculating the stress/shrieking level and the recovering procedure, the `StressDemon` class also provides the method `affectResources()`.
This method is intended model performance effects by turning stress into strain. It is called each time step, right before the resources are distributed between the demons. The `StressDemon` can implement this method by taking some resources, without providing behaviour for it in return. The idea is that the stress impairs resources, resulting in a strain on the agent. Other demons that depend on these resources have now less to work with, which may result in behaviour being executed poorly or not being executed at all.

3.6 Behaviour

An important part of the functionality of the `Demon` class in the original CHAOS framework is now outsourced to the abstract `Behaviour` class (Figure 11).
The two most important methods of `Behaviour` are `preOccupyResources()` and `takeAction()`. These methods are called from the `Pandemonium`, in each time-step. In the default implementation, the `Behaviour` class simply forwards these calls to the current `IBehaviourState`, that will be described in Section 3.7.
Subclasses can also choose to not use states, by overriding the methods `preOccupyResources()` and `takeAction()`. This is a good option if the `Behaviour` component is not very complicated and there is no need to divide its functionality in separate states.
A subclass that overrides these methods should take the resources it requires when `preOccupyResources()` is called. The retrieved resources can be stored internally by calling `storeResource(String,int)`. When the resources are needed later on, they can be retrieved by calling the method `getResourceStorage()`, that returns the whole collection of stored resources, or `getStoredResourceLevel(String)`, that only returns the stored level of a specific resource.

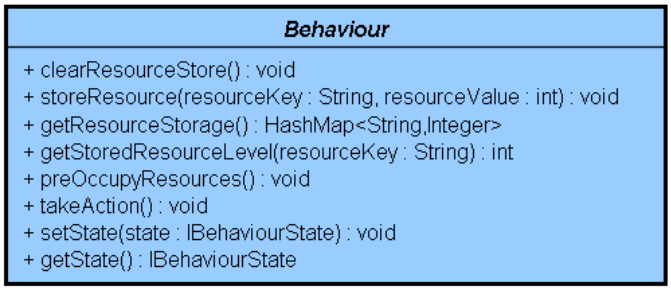

Figure 11 The `Behaviour` class.

A subclass that does not have an `IBehaviourState` associated should also implements the `takeAction()` method, that allows the `Behaviour` component to control the agent's behaviour.

3.7 IBehaviourState

The state pattern (Gamma, Helm, Johnson, & Vlissides, 1995) (Grand, 2002) allows specific functionality to be associated with specific states. This pattern is also used in CHAOS 2.0. for `Behaviour` components, that can be associated with an `IBehaviourState` (Figure 12). The Behaviour component can then simply forward calls of `preOccupyResources()` and `takeAction()`to the equivalent methods of its current state. Note that this is also the default implementation of `Behaviour`.
For instance, assume a `Behaviour` component has two potential states: `Paused` and `InProgress`. If the `Behaviour` is in state `Paused`, then calling `preOccupyResources()` and `takeAction()`would probably have no effect, since the `Paused` state would provide empty implementations for these methods. When the state of the `Behaviour` is changed to `InProgress` however, the resulting behaviour will change: resources will be taken and the behaviour of the agent will be influenced by the `Behaviour` component. Note however that, from the perspective of the `Behaviour` component, not much has changed as it is still just forwarding the `preOccupyResources()` and `takeAction()` method calls to its state.

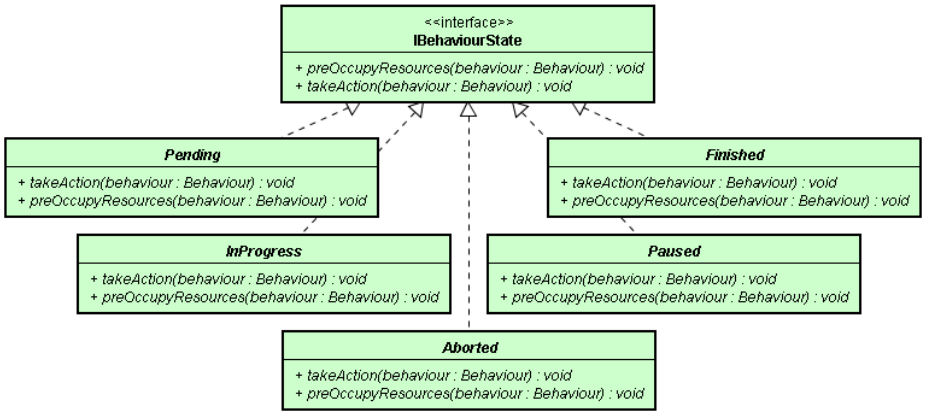

Figure 12 IBehaviourState interface and implementing subclasses.

The CHAOS library provides some default implementations of the `IBehaviourState` interface (see Figure 12). Since the CHAOS library is a generic library that provides no actual behaviour models, these implementations are abstract classes that provide no functionality. They are only provided for inspirational purposes and should be subclassed and extended with useful functionality.

4 Getting started with CHAOS 2.0

This chapter provides directions on how to develop behaviour models with the CHAOS framework. However, the reader should not expect a step-by-step recipe that can be followed precisely and then results in a perfect behaviour model. Behaviour modelling is too complicated for such a recipe to exist. There is no single best way of developing a behaviour model: each model is different and each domain and application have their own requirements and implications. For each domain and application, there are always many modelling solutions possible. In the end, it is up to the model builder to decide which solution is most suitable and how the model should be structured.
Although this chapter does not provide the ultimate recipe for behaviour models, it does describe the *structure* of such a recipe. In other words, it describes the steps that are involved when building a behaviour model, with pointers on how to execute each step. The chapter starts with a section on the high-level design of the behaviour model and continues with the actual implementation of the components and the interaction with a simulation engine. The text is illustrated with examples from the domain of traffic behaviour. These examples are printed in blue text blocks, such as the one below.

> *The example we shall be using in this chapter is from the traffic domain: the modelling of **driver behaviour** for a traffic simulation. For the sake of simplicity we shall only consider the behaviour of a person who is driving behind a lead vehicle on a straight road. The road may contain other vehicles, but does not contain crossroads or bends. We shall not include overtaking behaviour or non-driving related behaviours, such as using a mobile phone or the satellite navigation system.*

4.1 High level design

Before we can start implementing a behaviour model, we need to create a high level design on paper that describes the behaviours and resources. The following sections describe the most important steps involved in this process.

4.1.1 Determining the scope of the behaviour model

The first step in developing a behaviour model is to determine the scope of the model: which behaviours are relevant to the simulation and should be included? The answer to this question depends largely on the domain and type of application. Developing a traffic simulation aimed at research requires other behaviour models than a training simulation for the defence domain.
Note that additional behaviours can always be added to the simulation in a later stage. This is one of the advantages of the distributed modelling approach of CHAOS, with demons that can be viewed as “behaviour building blocks”. However, it still is helpful in this stage of the design to we have a global idea of the behaviours that should be included.

We want to create an agent that is capable of driving on a straight road, preferably without crashing into predecessors. A short analysis shows that this can be modelled with three different types of behaviour:

1. ***Free Driving**: the agent has to be able to drive at a preferred speed when no predecessors are nearby.*
2. ***Car Following:** the agent also has to be able to follow a predecessor at a suitable distance, when the predecessor is driving at a slower speed than the agent's preferred speed.*
3. ***Collision Prevention**: the agent has to be able to react quickly by braking, when its predecessor brakes hard or is driving much slower than the agent.*

4.1.2 *Proactive or reactive?*

Once it is clear which behaviours should be included, the next step is to divide the behaviours in proactive, top-down behaviours and reactive, bottom-up behaviours. Proactive behaviours are usually goal related, i.e. behaviours that help the agent to achieve its goals. Proactive behaviours are always initiated by the agent and, if nothing else happens, are all the agent will do. However, sometimes events occur that require the agent to react. These behaviours are reactive, bottom-up behaviours. These are triggered not by the agent but by an (external) event.

*The **Free Driving** behaviour is a typical example of proactive behaviour. It is the behaviour the agent will **act out** when no other traffic is on the road, or when the nearest predecessor is at a safe distance.*
*The **Collision Prevention** behaviour on the other hand is a typical example of reactive behaviour. A predecessor suddenly brakes hard; to prevent crashing into the predecessor's vehicle, the agent has to **react** by braking as well.*
*The **Car Following** behaviour is somewhere in between: it has to do with the goal the agent wants to achieve (get from A to B) but also involves reacting to the predecessor. We will for now assume it is part of the proactive behaviour of the agent.*

4.1.3 *Demons or behaviours?*

When it is decided which behaviours should be included and these are divided in proactive and reactive behaviours, the next step is to decide how these behaviours should be structured. As was described in Chapter 3, CHAOS 2.0 makes a distinction between *demons* and *behaviours*. This distinction is based on the strategy pattern (Gamma, Helm, Johnson, & Vlissides, 1995) (Grand, 2002), with the demon playing the role of *client* and the behaviour component acting as the *strategy* component.
A way to think of this distinction in the case of CHAOS is that the demon represents the *intention* of the behaviour (what is it that the agent wants to achieve) while the behaviour(s) represent the *realization* of this intention (how is the goal achieved). Another way to look at it is that the behaviour components form the "modes of operation" of the demon.

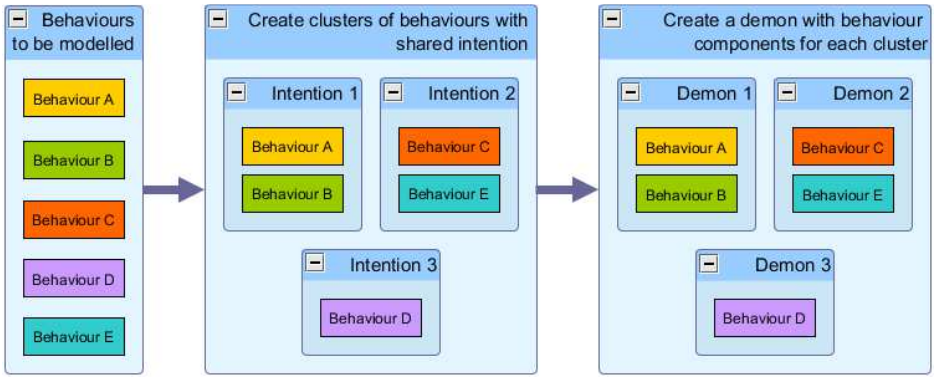

Figure 13 From a collection of behaviours (left) to a structure with demons and behaviour components (right). See also the example in the box below.

To determine which demons and behaviour should be created, the collection of behaviours that need to be modelled (see Section 4.1) should be clustered, with each cluster representing a specific intention of the agent. The behaviours in the clusters can be viewed as ways to achieve the goals that are associated with that intention. Once the clusters are identified, these can be translated directly to demons with behaviour components, as depicted in Figure 13.

> *We already identified three behaviours that need to be modelled: free driving, car following and collision prevention. The first two can be seen as belonging to "standard driving", or the intention of "getting from A to B", while collision prevention can be seen as belonging to the intention "Not crashing" or "staying alive".*
> *This would then call for two demons, that we may call* `Drive` *and* `DontCrash`*. The* `Drive` *demon gets two behaviour components, or modes of operation:* `CarFollow` *and* `FreeDrive`*. The* `DontCrash` *demon only has a single behaviour component that we call* `BrakeForPredecessor`*.*

4.1.4 *Which resources?*

When we have a clear view of the behaviours that are to be modelled, the next step is to think about the resources that need to be defined. A good starting point here is to think about the function a resource should play in the behaviour model. There are only two possible functions. The first function is to prevent that two conflicting, mutually exclusive, behaviours are executed simultaneously, e.g. walking and riding a bike. The second function is to modulate task performance, e.g. cognitive task performance as a function of an attentional resource. Note that performance can be modulated for several reasons, either because a resource is used by multiple behaviours (in the case of multi-tasking), or because a resource is impaired as a result of a stressor.

When we look at our traffic simulation example again, we can see that all the behaviour components, `CarFollow`, `FreeDrive` and `BrakeForPredecessor` are mutually exclusive: none of these behaviours should be executed simultaneously. This means that there must be a (set of) resource(s) that prevents simultaneous execution. In this case, a resource "RightFoot" may be sufficient to prevent simultaneous execution.

4.2 Implementation

When a global design of the demons, behaviours and resources exists, the next step is to start building the behaviour model in the CHAOS architecture.
This involves implementing demons and behaviour components, creating a pandemonium and resources, and linking the pandemonium and demons to an agent in the simulation. Optionally, some of the GUI components provided with the CHAOS library can be used to be able to inspect what is going on inside of a pandemonium. All these steps will be described in the following sections.

4.2.1 Creating demons

The CHAOS library does not contain demons or behaviour components that are ready to use, because it is impossible to know beforehand which behaviours are required. Furthermore, it cannot be known how the interaction with a simulation environment occurs. Therefore, for each application that uses the CHAOS library, custom demon and behaviour components need to be developed.
The CHAOS library does provide abstract classes that already contain a lot of functionality. To create custom demons and behaviours, the model builder "only" has to extend these abstract classes and provide the missing functionality.
The advantage of this approach is that functionality that is specific to a simulation can easily be added to the demon and behaviour objects. For instance, a demon could be given access to the simulation world, either to perceive its state or to manipulate it in some way.

A demon for a traffic simulation could be given access to data related to the roads, road signs, traffic lights, vehicles, etc. in the simulation. It could also be given access to the vehicle that is to be controlled by the agent, i.e. access to the steering wheel, brake and accelerator pedals.

Before implementing a demon, it needs to be decided which abstract demon class is to be used as the basis for the new demon. As Figure 8 shows, there are three options here: `ProactiveDemon`, `ReactiveDemon` and `StressDemon`.
For demons that represent proactive behaviour the `ProactiveDemon` should of course be used as a starting point. For demons that represent reactive behaviour, either `ReactiveDemon` or `StressDemon` can be used.
Whichever class is chosen, each demon should provide an implementation of the `updateBehaviour()` method. This method is called every time-step (i.e. each time `Pandemonium.update()` is called), *prior* to the execution of behaviour.
It gives the demon a chance to change its `Behaviour` component, or to provide new data or give new instructions to its `Behaviour` component.

An example of the `updateBehaviour()` method implementation for the `Drive` demon. Depending on "some condition", that is probably related to the distance to and speed of the predecessor, the behaviour component is set, either to "car following" or to "free driving":

```
public void updateBehaviour() {

  if(someConditionHolds) {
     setBehaviour(mCarFollowBehaviour);
  }

  else {
     setBehaviour(mFreeDriveBehaviour);
  }
}
```

Besides the `updateBehaviour()` method, the demons should implement methods that are related to their shrieking levels. Since the `ProactiveDemon` class already provides this functionality (i.e. a `setShriekingLevel()` method with `public` visibility), subclasses do not need to provide any additional functionality.
For the `ReactiveDemon` class, the `updateShriekingLevel()` needs to be implemented. In this method, the demon should determine its shrieking level and then call the `super.setShriekingLevel()` method with the new shrieking level as parameter. Note that the difference with `ProactiveDemon` is that this version of `setShriekingLevel()` has `protected` visibility, i.e. is only available to subclasses and package members.
If the shrieking level of the demon can be described as a linear function of a variable, as illustrated in Figure 9, then it may be useful to not extend `ReactiveDemon` directly, but use `StressDemon` instead. In that case, only a minimum and maximum and threshold value of the variable need to be set (see Section 3.5.2), and the variable value needs to be provided by implementing the `getMonitoredValue()` method. Another method that needs to be implemented is `affectResources()`. Through this method, the demon can increase or decrease resource levels as a result of increasing or decreasing stress levels. This can be viewed as the transition of stress into strain. If this is not applicable, the `affectResources()`method can be given an empty implementation. An example of a `StressDemon` implementation is given in the example box below.
The `StressDemon` class also provides functionality that is related to recovering from stress, which requires some additional methods that need to be implemented in a custom demon implementation. For the sake of simplicity, this functionality is not further described here and the reader is referred to the Javadoc documentation provided with the software library.

An example of the `DontCrash` demon implementation. It is implemented as a subclass of `StressDemon`. It uses the calculated "Time To Collision" (TTC) as the monitored value. In its `updateBehaviour()` method, it instructs the `BrakeForPredecessor` behaviour component to try to realize a certain deceleration.
Note that, for the sake of simplicity, the methods related to stress recovery are not included here.

```
public class DontCrash extends StressDemon {

// construction
public DontCrash(Pandemonium pandemonium, Driver driver) {

   // general initialization code here

   // stress related: min, max, threshold and direction,
   // based on Time To Collision (TTC)
   setIsMaxStress(false);
   setThreshold(8);   // start shrieking if TTC < 8
   setMaximum(1);     // shriek loudest when TTC <= 1 second

   // set (only) behaviour component:
   setBehaviour(new BrakeForPredecessor(driver, this));
}

// no resources to affect
public void affectResources() {}

// calculates time to collision (TTC) under current
// circumstances
private double calcTimeToCollision() {
   // calculations here //
   return ttc;
}

// the monitored value = TTC
public double getMonitoredValue() {
   return calcTimeToCollision();
}

public void updateBehaviour() {
   // provide new data/instructions to behaviour component
   // here, for instance by telling the brakeforPredecessor
   // component how hard it should be braking
}
```

Regardless of the type of demon that is implemented, it is usually sensible to add another layer of abstract classes to the lineage, instead of directly extending `ProactiveDemon`, `ReactiveDemon` or `StressDemon`. In these abstract classes, generic functionality, that is specific to the simulation and that is useful to all demons in the simulation, can be implemented. An example of this approach is described in the following example box and is illustrated in Figure 15.

Since the `Drive` demon represents proactive behaviour, the ProactiveDemon should be used as a starting point. However, to allow some generic functionality to be implemented for all proactive demons at once, it makes sense to first add another abstract demon that we could call `TrafficSimProactiveDemon` (see Figure 14).
The `Drive` demon should then be derived from `TrafficSimProactiveDemon`.

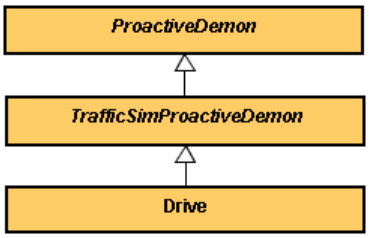

Figure 14 Example of a class diagram with an abstract class that contains simulation-specific functionality and that extends the more generic abstract `ProactiveDemon` class. The actual custom demon implementations, such as `Drive` in this example, are then derived from this abstract class.

4.2.2 Creating behaviours and behaviour states

Once the demons are created, we arrive at the "business end" of the behaviour model: the behaviour components. These behaviour components can be directly derived from the abstract `Behaviour` class, although it may be useful here as well to add an abstract class in between that can provide generic, simulation-specific functionality.

A behaviour component has two "tasks", the first is to try to collect the resources it requires, and the second is to influence the behaviour of the agent, given that sufficient resources are collected. This involves two methods:
the `preOccupyResources()` method and the `takeAction()` method.
The default implementation of the `Behaviour` class is that these method calls are forwarded to the current `IBehaviourState`, as follows:

```
public void takeAction() {
  if(getState()!=null) {
    getState().takeAction(this);
  }
}
```

The state itself can be changed by calling the `setState()` method. Which object manages the states and state changes (either the `Behaviour` object or the states themselves) is up to the model builder. More information on the state pattern in general can be found in (Gamma, Helm, Johnson, & Vlissides, 1995) and (Grand, 2002).

The model builder has two options: he can either provide `IBehaviourState` implementations that provide the required functionality, or he can override the `preOccupyResources()` and `takeAction()` methods in the `Behaviour` subclass, thereby effectively bypassing the state mechanism. It depends on the (complexity) of the behaviour which solution is preferred. Some behaviour can naturally be divided in different states, with different actions associated to each state. Other behaviours are simply active or not, in which case the state pattern may be unnecessary. An example of such a behaviour is given in the following text box.

The following code shows a possible implementation of the `BrakeForPredecessor` behaviour component. It tries to get access to the "RightFoot" resource and if it succeeds, changes the acceleration of the vehicle of the agent.

```
public void preOccupyResources() {
   storeResource("RightFoot",100);
}

public void takeAction() {
   if(getResourceStorage().containsKey("RightFoot")) {
      int value = getResourceStorage().get("RightFoot");
      if(value<100) return;// r.foot not available→do nothing
   }
   // right foot is available: do something with it:
   getDriver().getVehicle().setAcceleration(accel);
}
```

Whether the state pattern is used or not, resources will need to be taken and actions will need to be implemented. The example above illustrates the way to achieve this: taking a resource simply requires a call to `storeResource()`, with the key identifier of the resource and the amount specified as parameters. The `takeAction()` implementation in the example starts with checking if enough resources are available. This is a very important step and should always be included. If not enough resources are available, the `Behaviour` component can react in several ways. The simplest reaction is to just do nothing. Alternatively, a performance model could be implemented by changing the default behaviour or modulating the performance, in case not sufficient resources are available.

4.2.3 *Creating resources*

Once it is clear which resources need to be available to the demons, the actual creation of a `Resource` is rather simple and is achieved by a single constructor call. More information on the available constructors can be found in the Javadoc documentation that comes with the CHAOS library.

To create the "right foot" resource for our traffic simulation, the following constructor call suffices. It creates a resources called "RightFoot" that ranges from 0 to 100 and has an initial level of 0. Note that the resource still needs to be added to a pandemonium, which will be described below.

```
Resource rightFoot = new Resource("RightFoot",0);
```

4.2.4 *Using the ICHAOSAgent interface*

All agents that are to be controlled by CHAOS, i.e. by a pandemonium with demons, should implement the `ICHAOSAgent` interface. This interface provides some basic functionality to link the simulation and the CHAOS library together (see Section 3.2 for more information). All `ICHAOSAgent` objects should also be updated frequently, preferably from the main simulation loop, as is illustrated in the example in the box below.

To update the behaviour of the agents in the traffic simulation, each driver's pandemonium gets updated from the main simulation loop:

```
public void run() {

   Driver[] drivers = getDrivers();

   // this is the main simulation loop:
   while(true) {

      // update simulation here

      // update all simulated drivers, i.e. the ICHAOSAgents:
      for(ICHAOSAgent driver:drivers) {
         driver.getPandemonium().update();
      }

      // maybe do other things here
   }
}
```

4.2.5 *Creating and initializing a Pandemonium*

The creation and initialization of a `Pandemonium` is rather simple. The first step is to create (or get a handle of) the `ICHAOSAgent` that the pandemonium should be controlling. The next step is to call the `Pandemonium` constructor, with the agent object as parameter. After that, resources can be created and added to the pandemonium. The final step is to create the demons to add to the pandemonium. The following example block contains a code snippet to illustrate these steps.

The following code snippet is an example of how a driver can be created, with a pandemonium, demons and resources.

```
public void initDriver(String name) {

   // create an agent:
   Driver someDriver = new Driver(name);//implements ICHAOSAgent

   // create a pandemonium with a reference to the agent:
   Pandemonium pandemonium = new Pandemonium(someDriver);

   // create and add resources
   Resource rightFoot = new Resource("RightFoot",0);
   pandemonium.addResource(rightFoot);

   // create demons
    // (demons are added to the pandemonium automatically)
   new Drive(pandemonium);       // proactive "Drive" demon
    new DontCrash(pandemonium);     // reactive "DontCrash" demon

   // connect pandemonium and ICHAOSAgent
   someDriver.setPandemonium(pandemonium);
}
```

4.2.6 *Connecting GUI components to monitor the Pandemonium*

It is possible to inspect what is going on inside a `Pandemonium` at runtime, by making use of the GUI components that are provided in the CHAOS library. Two components are available for this purpose, the `DemonView` and `ResourceView` components (Figure 15).

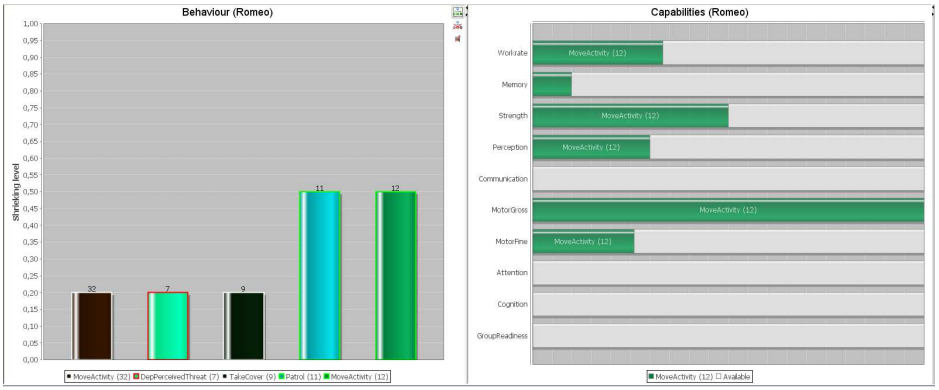

Figure 15 The `DemonView` (left) and `ResourceView` (right) components, taken from a screenshot of the SCOPE simulation.

The `DemonView` is used to inspect what demons are present and how loud they are shrieking. It represents the demons as bar charts, with the height of the bar representing the shrieking level. To use a `DemonView`, simply call any of the constructors to create it and add it to a container, such as a `JFrame`. The next step is to "connect" an agent with a `Pandemonium` to it. This can be achieved by calling `setAgent(ICHAOSAgent agent).`

The `ResourceView` component is very similar to the `DemonView`, but it shows resources instead of demons. The resources are represented as bar charts, with a colour coding that shows if and how much of the resources are used by which demons. In Figure 15 for example, the rightmost demon (`MoveActivity`) is the only demon using resources, as indicated by the green colour in the right `ResourceView` panel, that is the colour of the `MoveActivity` bar in the left panel of Figure 15. Using a `ResourceView` is identical to using a `DemonView`: after construction the view can be added to a container and an `ICHAOSAgent` can be associated by a call to `setAgent(ICHAOSAgent)`.

5 References

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software.* Reading, Massachusetts: Addision-Wesley.

Grand, M. (2002). *Patterns in Java* (2nd ed., Vol. I). Indianapolis, Indiana: Wiley Publishing.

Oracle. (2011). Retrieved January 2012, from The Java Tutorials: http://docs.oracle.com/javase/tutorial/index.html

Ubink, E., Aldershoff, F., & Lotens, W. (2008). *Models & Methods in SCOPE: A status report.* Soesterberg: TNO.

Ubink, E., Aldershoff, F., Lotens, W., & Woering, A. (2008). Behavior modeling through CHAOS for simulation of dismounted soldier operations. *Proceedings SPIE, Vol. 6965, 696504*.

Ubink, E., Lotens, W., & Woering, A. (2010). Towards a Generic Behaviour Modelling Interface. *Human Modelling for Military Application* (pp. 22/1-22/10). NATO RTO.

6 Signature

Soesterberg, March 2012

Drs. W.S.M. Piek
Head of department

Soesterberg

drs. E.M. Ubink
Author