RAM ROBOTICS AND MECHATRONICS

Design of animation facilities for analysing cyber-physical system software architectures

T.C. (Tjalling) Ran

MSc Report

September 2015

029RAM2015
Robotics and Mechatronics
EE-Math-CS
University of Twente
P.O. Box 217
7500 AE Enschede
The Netherlands

UNIVERSITY OF TWENTE.

MIRA BIOMEDICAL TECHNOLOGY AND TECHNICAL MEDICINE

CTIT

Summary

Design of cyber-physical systems is becoming increasingly difficult, due to increasing demands on functionality, as well as safety and time-to-market requirements. To efficiently manage this complexity, a design methodology, or 'way of working', has been developed.

The way of working is supported by the TERRA tool suite. Using TERRA's graphical editors, a cyber-physical system's software structure can be designed in a model-driven way. Through C++ code generation from TERRA, executables implementing the designed CSP model can be deployed on the target (embedded) system.

The execution flow of (the executable created from) the model is difficult to analyse, however. This hinders the desired iterative development. Additionally, it means that users new to TERRA and CSP have no efficient means to understand their own designs.

This thesis aims to address that issue, by adding animation facilities to TERRA. To establish requirements and planned features, existing animation tooling has been studied.

The execution flow of an executable created from a TERRA CSP model can be logged using already existing logging facilities. The animation facilities that have been realised, use the data provided by these logs to visually and textually depict the state changes in the model. As TERRA's editor is already graphical, the visual display of state changes can be implemented using the same representation. This ensures that users can easily understand the state changes they see in terms of their model. Furthermore, it is efficient in terms of implementation.

A textual view complements the graphical view, showing a history of state changes and describing them textually. Users can have multiple animations open at once, and step back and forth through the state changes at their own pace.

Animation results have been verified to be correct using CSP analysis of the models and analysis of data from the pre-existing logging facilities.

Planned functionality to add server capabilities to TERRRA for directly receiving log data has not been implemented due to time constraints. It is recommended to add this functionality in the future, as it would make animation significantly more user friendly.

During experiments, it was observed that the number of transitions taken is quite high, due to the fact that the CSP model elements each go through a fixed sequence of 3–5 states. Combined with a large amount of scheduling freedom due to CSP's *Parallel* constructs, this results in a large state space. Focussing on specific states is therefore somewhat difficult. This could be alleviated in the future by providing options to hide states, effectively reducing the size of the fixed sequence for each model element.

Additionally, an important recommendation is adding support for TERRA's Architecture models and external models. Currently, only CSP models can be animated. Adding this support would allow animation of more complex models that can, for example, contain communication with external hardware.

Contents

1 Introduction

1.1 Context

Cyber-Physical Systems (CPS) are becoming increasingly important in areas such as healthcare, industry and in people's homes. CPS are systems that have *physical* (often mechanical) functionality, which are controlled by the *cyber* part: one or more (often embedded) computer systems or microcontrollers.

There is an increasing demand in terms of functionality, which makes CPS increasingly more complex. In order to effectively manage this complexity, a design methodology, or 'way of working' for CPS has been developed (Bezemer, 2013). The way of working extensively uses Model-Driven Development (MDD) with (co-)simulation of these models and highly iterative development.

The TERRA tool suite (Bezemer et al., 2012; Bezemer, 2013) supports this design methodology, offering graphical Model-Driven Development of the software architecture, using (among others) a graphical representation of Communicating Sequential Processes (CSP) (Hoare, 1985) as a design language. Functional components from other tooling, e.g. 20-sim (Controllab Products, 2015), can be integrated in this design.

Implementation and deployment of the modelled software structure is supported through C++ code generation. Building the generated code results in an executable that can be deployed on the target (embedded) system. The execution flow of the executable adheres to the designed model.

For brevity, 'execution flow' is used hereafter to refer to the execution flow that would be obtained when generating code from a model, building that code and executing it.

1.2 Problem description

The most-used model type in TERRA is the 'CSP model', based on the process algebra 'Communicating Sequential Processes' (Hoare, 1985). A CSP meta-model has been developed (Bezemer et al., 2012), which is used by TERRA to define its CSP models. TERRA's CSP models are designed using its graphical CSP editor. TERRA is extensively used in our group's Master's course "Real-Time Software Development" (RTSD). The course is the students' first acquaintance with CSP, and our experience in teaching the course shows that students have difficulties understanding some of the modelling language's concepts.

A primary example is CSP's rendez-vous communication. Communication influences the execution flow of the model (intentionally), since it requires synchronisation between the transmitting and receiving ends. Analysis can quickly become complicated, especially for students and new developers that have no experience with the concept or implications of rendez-vous communication. Providing insight into a model's execution flow, clearly indicating synchronisation through rendez-vous communication, could therefore be highly beneficial for the learning process.

More generally, the design methodology for cyber-physical systems prescribes iterative development, which requires testing often. This means that analysing the execution flow is essential. Furthermore, frequent verification, and eventually validation, are required to ensure that the model's behaviour (at the interface level) does not change between the design methodology's steps.

From the above, it is concluded that insight into the execution flow is essential for effective modelling. TERRA facilitates this to some extent, by providing code generation of machine-readable CSP. The generated code can be analysed using tools such as FDR3 (Gibson-Robinson et al., 2014). However, a detailed execution flow analysis using these tools is unwieldy for all but the smallest models. The alternative, analysis of the built (deployable) executable, is difficult as it relies on console output, requiring manual code additions (e.g. `printf` statements) and / or knowledge of the underlying software framework, LUNA (Bezemer et al., 2011). This thesis aims to address the lack of insight, as discussed in Section 1.3.

1.3 Goal

This thesis' goal is to improve insight into the execution flow, by adding 'animation' facilities to the TERRA tool suite. These animation facilities depict execution flow graphically (and in part textually) in TERRA. The user can run or step through the execution flow manually, while receiving graphical feedback of the state changes in the model. Inspecting the execution flow this way allows the user to analyse his model efficiently, improving understanding and saving time and effort otherwise spent on low level debugging and learning the internals of LUNA.

Definition of Animation. Animation as used in this thesis is a form of 'software visualisation', or, more specifically, a graphical 'program animation'. Noble and Groves (1992) give the following definition of program animation:

> "*Program animation is the use of computer graphics and animation techniques to visualise the behaviour of an executing program.*"

In practice, this usually means providing a visual representation of stepping through the code itself, i.e. runtime debugging present in most major IDEs. For TERRA, with its more high-level graphical modelling, this translates to stepping through the state changes of the CSP model elements and visualising these changes.

1.4 Approach

LUNA's real-time logging functionalities (Wilterdink, 2011) are used to collect and transmit execution flow information. These functionalities define discrete states for all CSP constructs. Execution flow is defined as the state changes that occur, and their ordering.

Since models in TERRA are graphically represented and designed, the animation can reuse the existing representation to depict the state changes of all model elements.

To establish requirements, related tooling is studied, experiences gained in the RTSD course is used and architectural considerations of TERRA and LUNA are taken into account. Based on the requirements, the software architecture and user interface are designed and implemented. Correctness of the implementation is then demonstrated using tests and requirement coverage.

1.5 Document outline

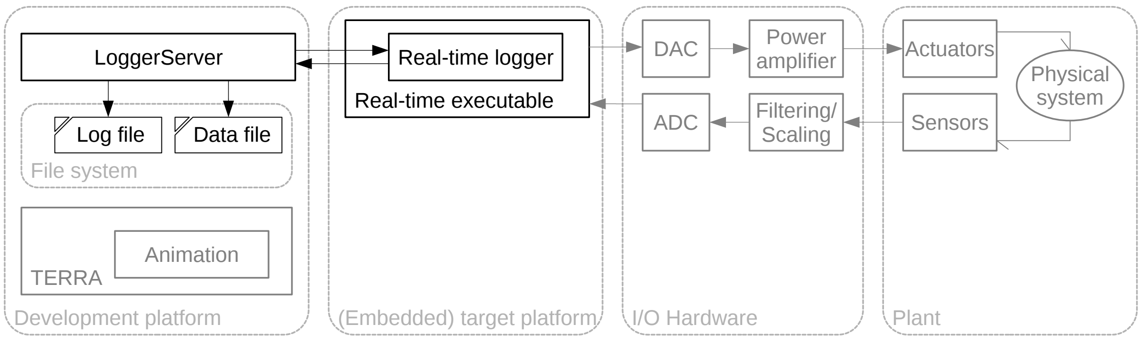
Requirements and the design of the animation facilities are presented in Chapter 2. Experiments used to verify the correct working of the animation facilities and their results are discussed in Chapter 3. Lastly, conclusions and recommendations for future work are given in Chapter 4.

2 Design

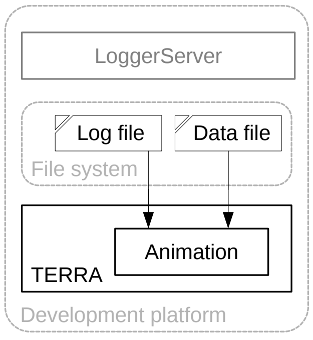
The design of animation facilities is treated in this chapter. Firstly, a typical setup in which animation would be used is defined in Section 2.1, to make the notion of using animation during development more concrete. Next, requirements for the animation facilities are given in Section 2.2. Per the requirements, the animation facilities have been designed in a modular manner. The corresponding high level software architecture is presented in Section 2.3. The architecture can be divided into functional blocks that each implement a role. The design of these functional blocks is discussed in Sections 2.4 to 2.7.

2.1 System overview

In a typical development setup, TERRA is run on a development machine, the software developed using TERRA is run on the target. This target platform is typically a hard real-time embedded system controlling a physical system, as shown in Figure 2.1. The figure is based on LUNA's pre-existing logging facilities: a real-time logger is part of the developed executable, and a standalone 'loggerserver' is used to retrieve and store the log data on disk. Using this setup, TERRA animation uses the log data on disk to animate its CSP models. This is discussed in more detail below.



(a) Collection of execution flow data



(b) Animation using the collected execution flow data

Figure 2.1: A control system of which the software structure is animated in TERRA, using offline logging

All active components relevant to animation are shown in black in Figure 2.1, other components in grey. The figure depicts 'offline' animation. In offline animation, all execution flow data is collected and saved in log files (Figure 2.1a). Afterwards, the log files are imported in TERRA and used to animate the corresponding TERRA CSP model (Figure 2.1b).

The physical system, or 'plant', is shown on the right-hand side of the setup, together with the actuators (e.g. motors) used to steer it and the sensors (e.g. encoders) that provide feedback to the control system.

Via Input/Output (I/O) hardware, these sensors and actuators are connected to the real-time executable. The real-time executable, shown running on an (embedded) target platform in Figure 2.1a, controls the plant. The executable has been developed through modelling its software structure in TERRA (and, e.g., integrating the controller developed in 20-sim), as described in Section 1.1.

To analyse the behaviour of the CSP model elements' implementation in the executable, the executable includes a real-time logger. This logger transmits its log data to a server program, the 'loggerserver', running on the development platform (shown in the top left of Figure 2.1a). The loggerserver writes the log data to files on the file system.

Once a sufficient amount of log data has been received, the system can be turned off. Next, the log data is imported in TERRA (Figure 2.1b) and used to show the execution flow to the user.

Figure 2.1 shows a standalone loggerserver. As an alternative to the setup of Figure 2.1, such server functionality can be implemented in TERRA's animation facilities, as shown in Figure 2.2. That way, the logged data can directly be used for animation. This approach could have support for 'online' animation, in which the executable can be animated while running. While clearly the superior alternative from a user's perspective, it does require additional implementation effort compared to using the pre-existing loggerserver.

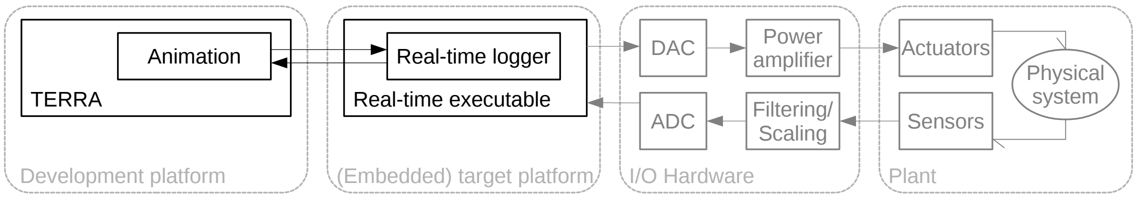


Figure 2.2: A control system of which the software structure is animated in TERRA, using online logging

Figures 2.1 and 2.2 show the presence of I/O and plant. In reality, these are often not present in the early stages of design, when focus is on the software structure, i.e. the TERRA CSP model, itself. The model is (usually) independent of parameter values and of algorithms (such as 20-sim control algorithms) embedded in it. Therefore, the executable can be run as is, on the target or on the development machine itself. This allows for quick analysis of the execution flow. A schematic showing the executable being run on the development machine is given in Section 3.1.

2.2 Requirements

This section lists the thesis' requirements. These have been established using:

- Knowledge of the system architecture (see Section 2.1) and TERRA and LUNA
- Evaluation of features from related work and tooling:
  - *gCSP's animation facilities* (van der Steen, 2008). gCSP is the predecessor of TERRA. Studying gCSP mainly led to insights pertaining to a graphical animation view, progressing through animations, and customisable graphical feedback.
  - *IBM Rational Rhapsody Developer* (IBM, 2015). Rational Rhapsody Developer features animation of UML state machine diagrams. Studying this animation mainly resulted in insights for a graphical animation view.

The requirements are given and discussed in Sections 2.2.1 to 2.2.4. They are prioritised according to the MoSCoW method (Clegg and Barker, 1994).

2.2.1 General requirements

Requirement 1: *Animation* must *be possible, with support for starting, stepping, pausing and stopping.*

The user must be able to follow the CSP executable's execution flow. This requires facilities for showing the execution flow and progressing through it. Requirement 2 and / or Requirement 3 is necessary to satisfy this requirement.

Requirement 2: *Offline animation* should *be possible*

Execution flow data can be stored in log files by LUNA's loggerserver. It should be possible to import these log files in TERRA and animate using the logged data.

Requirement 3: *Online animation* should *be possible.*

Animating during a run of the executable allows for a more interactive analysis. For example, the user could have his model request input during execution, allowing him to influence the execution flow (e.g. using CSP's Alternative construct). Performing such activities in an online run can significantly reduce development overhead compared to offline animation.

2.2.2 Animation view requirements

Requirement 4: *The animation* must *include a graphical model view that visually represents (changes in) state.*

A graphical view is best suited to display a state configuration at a certain point in the executable's execution. The state configuration is the aggregate result of all state changes from the start of the execution up to that point.

Requirement 5: *The animation* must *include a textual view, showing execution history.*

A textual view can complement the graphical view. Firstly, a textual description of state changes can add detail to the graphical changes. Secondly, it can list all preceding state changes, of which the graphical view shows the aggregate result (but not how that result came to be). In other words, a textual view shows history.

Requirement 6: *The user* could *have the option to customise graphical and textual feedback*

The depiction of state changes should be easy to interpret by the user. Such depictions, colour association in particular, are subjective. Therefore, customisation options are useful.

2.2.3 Communication requirements

Requirement 7: *TERRA* should *be able to retrieve log data directly from the RTLogger, without using the RTLogger-server*

TERRA should have server functionality (instead of LUNA's loggerserver) to receive execution flow data from the real-time logger. This improves integration and ease of use, and is important for online animation (see Requirement 3).

Requirement 8: *The communication facilities* should *be platform independent*

Executables resulting from development in TERRA are run on multiple target and development platforms. Therefore, communication of the log data should be implemented in a platform independent manner.

Requirement 9: *The impact of data loss* could *be minimised.*

Loss of execution flow data can occur. One example is unlogged data due to real-time constraints, since LUNA's real-time logger is assigned a lower priority than the CSP model part of the executable. Another is data loss due to network issues. TERRA's animation facilities could be designed to cope with this (to an extent), without giving the user erroneous animation results.

2.2.4 Non-functional requirements

Requirement 10: *Use of the animation facilities in the modelling suite* should *be intuitive*

Easy to learn and use animation facilities are important for user experience and adoption. The user should be able to focus on understanding his model during animation, not have difficulties with the animation facilities themselves.

Requirement 11: *The animation facilities* must *be modular with respect to the rest of TERRA and be architecturally comparable.*

TERRA is designed in a modular fashion, as is Eclipse. The animation facilities must adhere to this design philosophy.

Requirement 12: *The animation facilities* must *not compromise the integrity of the original TERRA CSP model*

Animation uses the model, and a view thereof, to analyse execution flow data and depict sate changes. All access to the model should be read-only, as the model itself must not be changed.

2.3 Architecture

The Animation functionality can be divided into functional components, each responsible for a specific role. This division into components is translated to a division into Eclipse plugins, resulting in the architecture shown in Figure 2.3. As shown in the figure, an additional division is made, into a CSP-specific implementation and a more generic 'base' implementation (or specification). This provides a basis for implementing animation of other model types in the future. For example, support for TERRA Architecture models could be added, as illustrated by the greyed out, hypothetical plugins in Figure 2.3.

Full plugin names include the common prefix of TERRA's eclipse plugins, `nl.utwente.ce.terra.`. For brevity, this prefix is omitted in the following. For example, the base animation plugin's full name is `nl.utwente.ce.terra.anim.base`, which is shortened to `anim.base`.

The roles and corresponding plugins are as follows:

- **Core functionality (**`anim.base, anim.csp, anim.csp.luna`**).** Core animation functionality includes starting and stopping animations, and interpreting execution data. The interpreted data is exposed to 'clients', who also are also given acces to means to step through 'an animation'. This core functionality is discussed in Section 2.4.

  In the above and hereafter, 'an animation' refers to an animation session, 'stepping through an animation' means progressing through its state changes. 'Clients' are objects (in the programming sense) interested in other objects or the data they expose.

- **User interface (**`anim.base.ui, anim.csp.ui`**).** User interface functionality that is not related to a specific animation view. This includes starting animations and stepping through them. Hence, it provides a front end to functionality defined in `anim.base` and `anim.csp`. This role is discussed in Section 2.5.

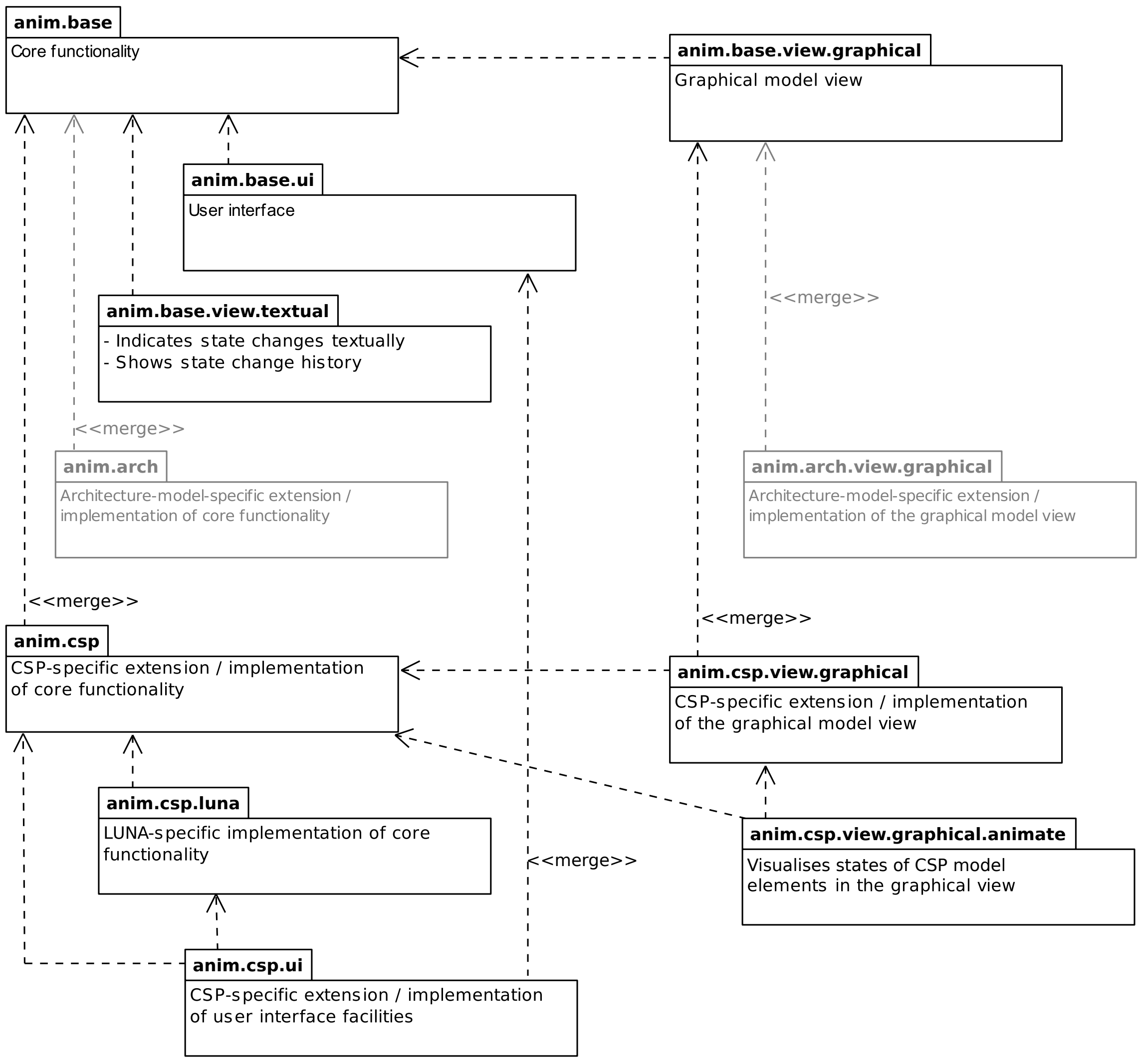

anim.base
Core functionality
anim.base.view.graphical
Graphical model view
anim.base.ui
User interface
anim.base.view.textual
- Indicates state changes textually
- Shows state change history
<<merge>>
anim.arch
Architecture-model-specific extension / implementation of core functionality
anim.arch.view.graphical
Architecture-model-specific extension / implementation of the graphical model view
anim.csp
CSP-specific extension / implementation of core functionality
anim.csp.view.graphical
CSP-specific extension / implementation of the graphical model view
anim.csp.luna
LUNA-specific implementation of core functionality
anim.csp.view.graphical.animate
Visualises states of CSP model elements in the graphical view
anim.csp.ui
CSP-specific extension / implementation of user interface facilities

Figure 2.3: Plugin diagram of the TERRA Animation facilities. Hypothetical future additions are greyed out.

- **Graphical animation view (** `anim.base.view.graphical`, `anim.csp.view.graphical`, `anim.csp.view.graphical.animate`**).** The graphical representation of state changes. Discussed in Section 2.6.
- **Textual animation view (**`anim.base.view.textual`**).** Textual representation of state changes. Discussed in Section 2.7.

There are a number of design decisions, related to Model-Driven Development, that are relevant to the architecture discussed in this section. These are discussed next, in Section 2.3.1.

2.3.1 Design alternatives

Storing execution data

As discussed above, part of the 'Core functionality' is exposing execution flow data to clients. As discussed in Chapter 1, the execution flow data, in the case of LUNA CSP executables, consists of state changes of the CSP model elements.

These state changes can be stored in multiple ways. One alternative is extending the existing TERRA CSP model elements with a state property. This would either require changes to the existing model itself, or defining a new model that inherits from the existing model. Since the states only have meaning in the context of animation, adding them in the existing model would not be good practice. Defining a new model is better option.

An additional issue is that previous and future state changes needs to be stored, to enable stepping through the animation. When using the model-based approach, implementing this range of state changes is not straightforward. Additionally, retrieving a set of state changes is difficult if the state changes are distributed among the model elements.

Therefore, an alternative that is not model-based has been chosen. The state changes are stored in a database object, along with references to the model elements they apply to. These model elements are those defined in the existing (CSP) model, without extending that model. The database is discussed further in Section 2.4.5.

Mapping logged model elements to TERRA model elements

As the execution flow data consists of state changes of model elements, the logs need to provide some way to indicate which model element a state change applies to. Combined with the fact that the executable implements the model, this means that the executable needs to have some notion of these model elements.

In LUNA this is indeed the case. All model elements are declared in the first part of the log. When interpreting the log data, these model elements need to be mapped to the corresponding elements in the TERRA CSP model, in order to determine to which model element a state change applies.

LUNA's logs give the model elements by their name, so naming information can be used to perform the mapping. However, TERRA allows names to be used more than once (except in the same diagram), which makes name-based mapping difficult.

A solution is given by LUNA: the model structure is defined as a hierarchical tree in LUNA. For example, the constructs contained in a Parallel group are that group's children, the group is their parent. All siblings do have mutually unique names, which eases name-based mapping.

One option of implementing the mapping is recreating the LUNA tree from the log, creating a similar tree from the TERRA model and comparing the resulting trees. This approach has been implemented, see Section 2.4.5.

Of course, the trees can be considered a specific type of representation of the model. This tree representation can in itself be a model as well. This would be a model-driven way of looking at the mapping. A well-defined tree structure meta-model could be useful in the future, for example for creating a graphical tree view in TERRA.

There is one issue, however: LUNA and TERRA's C++ LUNA code generation govern the tree structure as used in the executable. There are subtle hierarchical differences between that tree structure and a tree structure as it would be defined in TERRA, which presents problems. These differences could be resolved using a model-to-model transformation, to abstract away from LUNA specifics.

The tree structure meta-model is determined to be the best alternative. It was only considered at a state in development, however, where implementation was not feasible anymore. Instead, an approach fully realised in code has been used. This approach constructs a tree from the TERRA model in such a way that it adheres to the tree structure as it would be defined in LUNA.

Thus, for both the mapping and the storage of execution data, code-based solutions have been chosen. Having established this, the animation plugins do not have to take additional GEF models into account. The design of the plugins is discussed next, in Sections 2.4 to 2.7.

2.4 Core functionality

`anim.base, anim.csp, anim.csp.luna`

The core animation functionality can be subdivided into a number of responsibilities. These are given and discussed in Sections 2.4.1 to 2.4.7. The main classes representing these responsibilities in the `anim.base` plugin are shown in Figure 2.4. Interfaces for which no realisation is given in the figure are realised in the `anim.csp` or `anim.csp.luna` plugins.

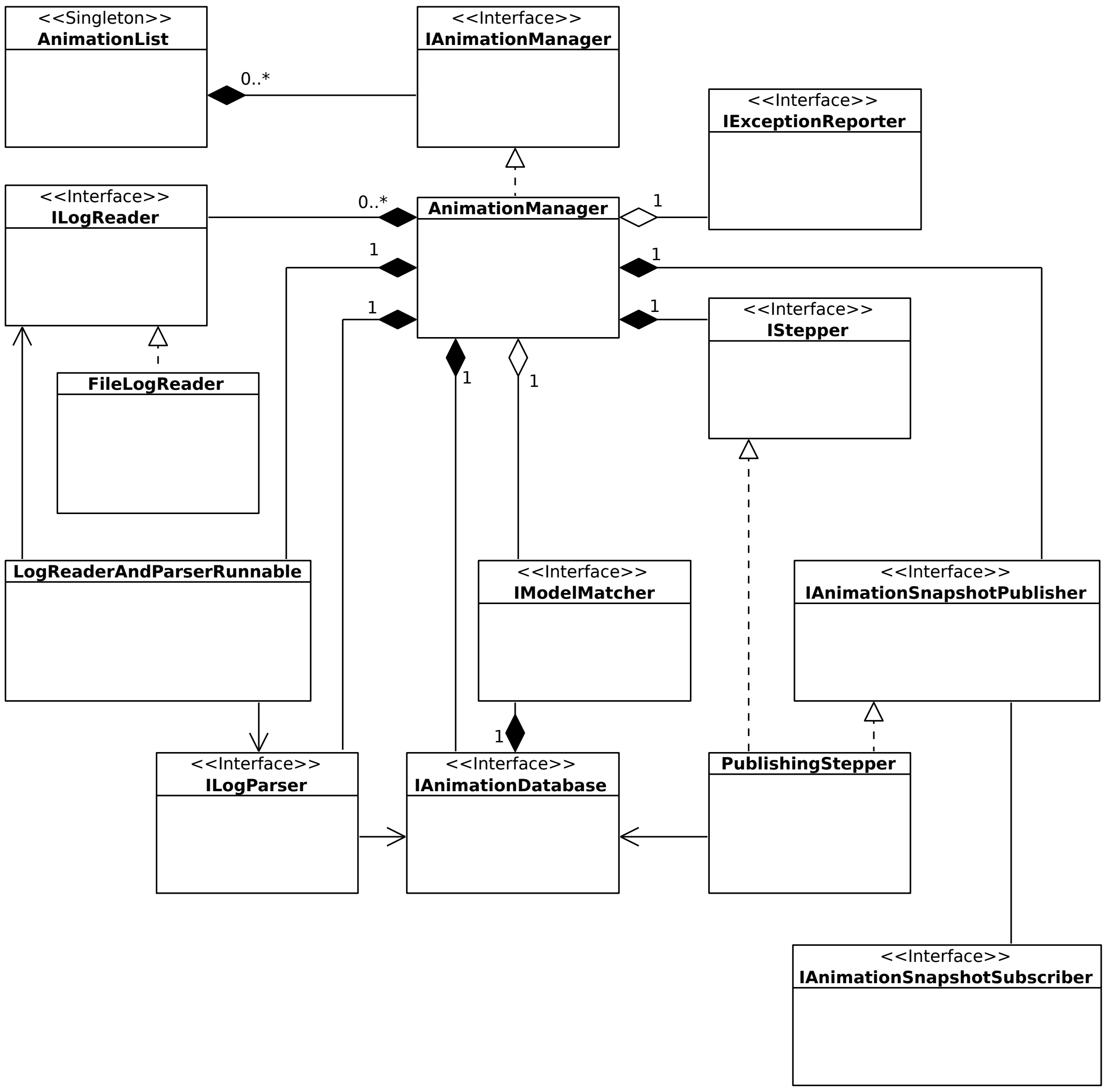


Figure 2.4: Class diagram depicting the core classes in the `anim.base` plugin

A concise, CSP-specific description of the core plugins' responsibilities is best illustrated by describing the creation and usage of a new animation:

1. An animation is created, and added to the list of currently active animations. Clients can obtain a reference to this animation via the list.
2. Logged execution data is imported by the animation. All CSP model elements found in the log are located in the TERRA CSP model that is to be animated, to provide a mapping between the log and the model. Next, all logged state changes of these model elements are stored. Each log entry containing state changes is assigned a 'step number' (or simply 'step').

3. Clients obtain a reference to the animation's `IStepper` object, which exposes facilities for 'stepping' through the animation, i.e. proceeding to the next step and its corresponding set of state changes. Clients that have subscribed to the animation's `IAnimationSnapshotPublisher` are notified of each step taking place, and are passed the corresponding state changes.

2.4.1 Handling and accessing multiple animations

`AnimationList`

The `AnimationList` acts as the entry point for clients, to obtain references to animations. It maintains a list of all currently existing animations and has facilities for 'focusing', creating, and terminating animations. An animation is focused if the user is currently interacting with that animation. Only a single animation can be focused at a time.

Clients can either obtain a reference to the currently focused animation, or through the animation's unique numeric ID, which is assigned by the `AnimationList`. An alternative means of identification would be the TERRA CSP model. However, that would imply that only a single animation per model can be active at a time. This is undesirable, since comparing two different runs of the same model's executable is a valid usage scenario.

`AnimationList` is a singleton, for two reasons: all animations are maintained in a single list to manage focus, and easy access by clients is essential because of its entry point role.

2.4.2 Creating and managing an animation's core functionality

`IAnimationManager, AnimationManager`

`IAnimationManager` is responsible for the components that together form the core functionality of an individual animation: obtaining and parsing execution data, coupling it to the model under animation, and stepping through the animation. These components are discussed in Sections 2.4.3 to 2.4.6.

The `IAnimationManager` exposes the components that clients are allowed to interact with, providing them facilities to step through the animation and subscribe to state changes.

The `anim.base` plugin provides a default implementation, `AnimationManager` (see Figure 2.4). For instantiation of its core components, `AnimationManager` delegates to a factory. This ensures that it does not need to know implementation details of the corresponding classes, only their interfaces. This allows `AnimationManager` to be implemented in the generic `anim.base` plugin, while specific implementations (and their factories) are located in specialised plugins.

Reading and parsing logs should be done asynchronously with respect to creating and stepping through the animation. Therefore, `AnimationManager` spawns a thread with a `LogReaderAndParserRunnable` (see Figure 2.4) that takes care of these tasks. Similarly, continuous stepping (i.e. running) should not be performed on the main thread. This is discussed in Section 2.4.6.

2.4.3 Obtaining execution data

`ILogReader, FileLogReader`

The execution data logged by LUNA's real-time logger (or potential future loggers) needs to be obtained by the animation. This is the `ILogReader`'s responsibility. It is a generic reader interface, providing a method to obtain a single log entry at a time. This allows it to be used for any type of input source and data type. `FileLogReader` is an implementation that uses text files as input source, where the text files are assumed to have a single entry per line. This is consistent with the output format of LUNA's loggerserver (by design).

2.4.4 Parsing execution data

`ILogParser, LUNACSPLogParser`

Once a log entry has been obtained, it needs to be parsed. This is the responsibility of `ILogParser`. Since the log format and contents are dependent on modelling language and logger implementation, no default implementation is provided in the `anim.base` plugin. The `anim.csp.luna` plugin provides an implementation specific to LUNA's loggerserver and CSP, `LUNACSPLogParser`.

The `LUNACSPLogParser` adds all model elements that are provided (one per log entry) in the log to the `IAnimationDatabase`, which matches them to the corresponding TERRA CSP model element. Next, each step (i.e. set of state changes) in the log is parsed and added to the database, which processes and stores it.

2.4.5 Coupling execution data to the model under animation

`IAnimationDatabase, AnimationDatabase IModelMatcher, ModelMatcher`

Parsed log data is offered to the database, as stated in Section 2.4.4. These additions to the database consist of two phases: a registration phase and a state change phase (or main phase).

The registration phase consists of registering all model elements declared in the logs with the database (an `IAnimationDatabase`). For CSP models, this includes all Processes, Readers, Writers, Recursions, Code blocks, and Compositional groups. These model elements are looked up in the TERRA (CSP) model under animation. The database delegates this lookup to its `IModelMatcher`.

The model matcher constructs a tree structure from the TERRA model and a second tree from the model elements from the log, as provided to it by the database, as discussed in Section 2.3.1. The trees are compared to find matches between model elements from the log and model elements in the TERRA mode.

If a matching model element is successfully found, a mapping is created between the ID used for that element in the log and the reference to the model element in TERRA. If no matching model element can be found, this means that the executable from which the log was created does not correspond to (the version of) the model under animation. Therefore, a mismatch is reported to the user (using the error reporting facilities discussed in Section 2.4.7), and the animation is prevented from continuing.

Once the full model element mapping is complete, updates to model elements' states can be added to the database, which marks the beginning of the the state change phase. In practice, these state updates are provided to the database by the log parser, see Section 2.4.4. In the CSP implementation of the database in `anim.csp` (see below), each new log entry adds a new step to the database. These steps are constructed by comparing the state configuration from the log entry to the prior state configuration in the database, creating a 'diff' of the state changes. These diffs, combined with metadata such as the step number, are stored in 'animation snapshots', i.e. `AnimationSnapshot` objects.

The database exposes these snapshots to clients. In practice, the client is an `IAnimationSnapshotPublisher`, which distributes the snapshots to its listeners. The `IAnimationSnapshotPublisher` is discussed in Section 2.4.6.

Since the database and the model element matching are dependent on modelling language, the `anim.base` plugin only provides interfaces: `IAnimationDatabase` and `IModelMatcher`. Implementations for TERRA CSP models are provided in the `anim.csp` plugin: `AnimationDatabase` and `ModelMatcher`.

2.4.6 Progressing through an animation

`IStepper, IAnimationSnapshotPublisher, PublishingStepper`

Progressing through an animation consists of two tasks. The first is the progressing itself, i.e. stepping. The second task is publishing the state changes that make up the steps to all interested parties (i.e. subscribers).

Progressing is performed by an `IStepper` object, which provides facilities for taking a single step (forwards or backwards), running up to a certain step, running continuously (until being paused or the last step is reached), and jumping to a specific step. Using Eclipse's Job API, running is offloaded to a worker thread, to ensure that Eclipse remains responsive.

Clients are informed of each step and the state changes therein, by the publisher. The publisher is an instance of `IAnimationSnapshotPublisher`. The information about a step is packaged as an `AnimationSnapshot` (see Section 2.4.5). Clients can subscribe themselves to these snapshots through the publisher. If they register after the first step has taken place, they can request all previous updates to obtain the full history.

As the tasks of progressing and updating are closely related, both interfaces are implemented in a single class, `PublishingStepper` in the `anim.base` plugin. Clients are unaware of this[1], as they obtain references to the object by one of its two interfaces (from the `IAnimationManager`).

2.4.7 Reporting problems

`IExceptionReporter`

As the UI and non-UI facilities are separated by design, the core animation functionality has no direct facilities for reporting problems to the user. An animation is always started by the user, and therefore from a UI plugin. This UI plugin can provide facilities for reporting problems. To allow the core functionality to use these facilities, an `IExceptionReporter` interface is defined in the `anim.base` plugin. This interface is implemented in the `anim.csp.ui` plugin (see Section 2.5).

In `anim.base`'s `AnimationManager`, constructor injection[2] is used to provide it with an exception reporter. The `AnimationManager` handles exceptions thrown by the objects it instantiates, and instructs the exception reporter to provide feedback to the user.

As the name implies, problem reporting is done through Java exceptions. One reason for this choice is that it allows using a single, well-defined system for reporting all problem types. Additionally, exceptions propagate up the call stack, until they are caught. This means that the classes throwing the exceptions do not need to know about any exception handling mechanisms.

An alternative is using methods that return error codes. Since methods can only return a single value, this means that either the error code and desired return value need to be wrapped in a struct-like object, or the method needs to be passed an object in which to store the value. Additionally, error checks and error propagation need to be manually implemented. For these reasons, exceptions are considered the superior option.

[1] Provided that they do not use Java's reflection facilities.

[2] With constructor injection, the constructor method requires a parameter (e.g. a class instance) as one of its arguments. Therefore, creation of that instance is delegated to code external to the constructor. In this case, it allows any class implementing `IExceptionReporter` to be used, without the `AnimationManager` knowing about the specific implementation.

2.5 User interface

`anim.base.ui, anim.csp.ui`

The UI plugins provide animation functionality that is not related to a specific view (such as the textual or graphical view). They provide a toolbar to focus animations and progress through them, context menu entries for starting animations, and an exception reporter (see Section 2.4.7).

The toolbar has a drop-down box for selecting which animation is currently focused, buttons for progressing through the focused animation and a button to terminate (close) the focused animation. It is discussed further in the separate user manual document. The toolbar has been implemented using Eclipse's extension points for menus, commands and handlers.

An "Animate" entry has been added to the context menus in TERRA's CSP Editor. This entry starts an animation of the CSP model that is open in the editor, after prompting the user for log files to be used for the animation. A similar context menu entry is shown when right-clicking on a TERRA CSP model (`.cspm` file) in Eclipse's project explorer.

2.6 Graphical animation view

`anim.base.view.graphical, anim.csp.view.graphical,`
`anim.csp.view.graphical.animate`

The graphical animation view derives from the TERRA editor. As such, the CSP graphical animation view reuses figure classes defined in the TERRA CSP editor. The view visualises model elements' states by formatting these figures, i.e. changing their colouring and border thickness.

The main changes with respect to the TERRA CSP editor are:

- **Read-only-ness.** The graphical view is read-only to prevent users from modifying the model during animation. This is discussed in Section 2.6.1.
- **Animation facilities.** Extensible facilities for visualising state (changes) have been added to the `anim.csp.view.graphical` plugin. These are discussed in Section 2.6.2.

2.6.1 Read-only-ness

The graphical view maintains TERRA's editor architecture and, therefore, that of the Graphical Editing Framework (GEF) (Rubel et al., 2011). GEF uses the Model-View-Controller design pattern (MVC) (Krasner and Pope, 1988). In GEF's implementation, `EditParts` are the controllers and are responsible for creating the view's figures and editing the (CSP) model. `EditParts` (mostly) delegate their tasks to `EditPolicy` objects. Therefore, the `EditPolicy` objects installed on an `EditPart` determine the `EditPart`'s capabilities.

The graphical animation view uses custom `EditParts` and `EditPolicy` objects for all model elements: the only policies that are installed are one that allows selection of figures in the view and policies used to visualise state. The `EditParts` ignore 'Direct Editing' requests to prevent in-editor renaming of model elements.

A drawback of this approach is that it requires all `EditParts` in TERRA's `csp.editor` plugin to be extended individually in the `anim.csp.view.graphical` plugin, each with (nearly) identical implementation changes. This negatively impacts maintainability. This is offset by the straightforwardness of the implementation, which adheres to standard practice for GEF editors.

One alternative solution is using `EditParts` that wrap the `EditParts` in TERRA's `csp.editor` plugin. However, this has the drawback of needing to wrap all of the original `Editparts`' methods, of which there are many. More significantly, it would interfere with code using `instanceof` checks. Therefore, this alternative has not been chosen.

A second alternative would be extending `EditPart`s with methods to remove all installed `EditPolicy` objects. In practice, this results in first installing all edit policies as used in TERRA's `csp.editor`, then removing all edit policies, and installing the edit policies used in animation. This is wasteful and considered to interfere with intended behaviour, which is why this alternative has not been chosen either.

Besides a read-only editor, Eclipse's properties view also needs to be read-only. Therefore, the graphical view uses a custom provider for Eclipse's properties view. Contrary to the CSP editor's provider, it creates property descriptors that do not allow properties to be edited, effectively making the properties view read-only.

2.6.2 Animation facilities

Edit policies

Visualisation of state changes is implemented through `EditPolicy` objects. The `anim.csp.view.graphical` plugin defines an extension point for adding edit policies. These edit policies are automatically installed on all edit parts, using a factory. This approach enables future extensions to the visualisation, by contributing additional edit policies. Currently, an `EditPolicy` is included which can handle requests to set figures' fore- and background colours and linewidth. This is both for animation purposes and an example for clients wishing to extend animation functionality.

The edit policies add capabilities to an edit part, by handling requests. In order to use these capabilities, a request must first be passed to the edit part. Generating these requests is discussed in section 'Animator' below. The edit part queries each of its edit policies whether they can handle the request type. The edit policies that can, do so.

Animator

To visualise state changes on a CSP model element's figure, requests to do so must be generated and passed to the corresponding edit part (see section 'Edit policies' above). An extension point to do so has been defined in the `anim.csp.view.graphical` plugin. In the following, implementers of this extension point are called 'animators'.

In the implementation used in this thesis, the `anim.csp.view.graphical.animator` plugin implements the extension point. The animator subscribes to the animation's `IAnimationSnapshotPublisher` (see Section 2.4.6) and generates visualisation requests based on the state changes it receives. Whenever it receives a new `AnimationSnapshot` from the publisher, it locates the `EditPart`s corresponding to the changed states, and has them visualise their new state by creating a corresponding request. The `EditPart`s delegate this request to their corresponding `EditPolicy`.

The extension point can be implemented by other plugins, which allows additional visualisation functionality in the future.

Figures

To fulfil the request, the `EditPolicy` requires figure formatting capabilities. The figures used by TERRA's `EditPart`s have been refactored (as high up the hierarchy as possible) to derive from the newly created `FormattedFigure` class. This class provides default fore- and background colours and allows its line width to be set.

The graphical view plugins add colour and line width settings to the Eclipse preferences, for each state. These settings define the formatting used by the `EditPolicy`.

2.7 Textual animation view

`anim.base.view.textual`

This view shows state changes textually, in a table. It complements the graphical view in two ways. Firstly, it clearly states the state changes, in a flat (i.e. non-hierarchical) representation. Secondly, the table shows all previous steps and can therefore be used to see the execution flow history without stepping back. The graphical view, on the other hand, shows the current state configuration.

The textual view has been implemented using Eclipse JFace's `TableViewer`, which uses an MVC approach. The `TableViewer` itself is the entry point, which delegates processing of its input to provider classes, `ContentProvider` and `LabelProvider`. The `ContentProvider` receives new animation state changes from a publisher and instructs the `TableViewer` to add this data to the table. Upon receiving this instruction, the `TableViewer` creates a new row (if necessary) and delegates filling its contents to its `LabelProvider`. This way, the `TableViewer` does not need to know about the underlying data model.

As no CSP-specific model information is required, the `anim.base.view.textual` plugin contains the full implementation.

3 Results

In this chapter, the animation functionality is verified to work correctly. Two conceptually similar experiments have been performed. Their setup is presented and motivated in Section 3.1. The individual experiments are presented in Section 3.2 and Section 3.3. Section 3.4 discusses their results and general observations.

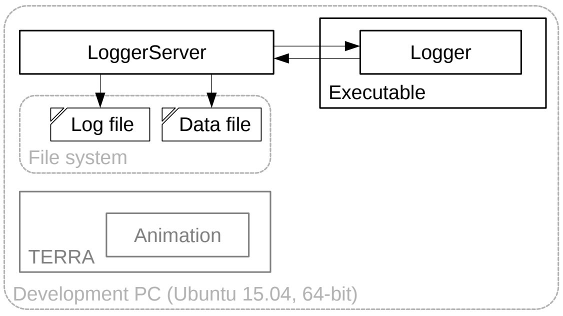
3.1 Experiment setup

The test procedure is based on visually verifying correctness of the animation, using the graphical and textual animation views. This automatically tests the core functionality and the UI facilities, as these are required for the views to display the correct state changes.

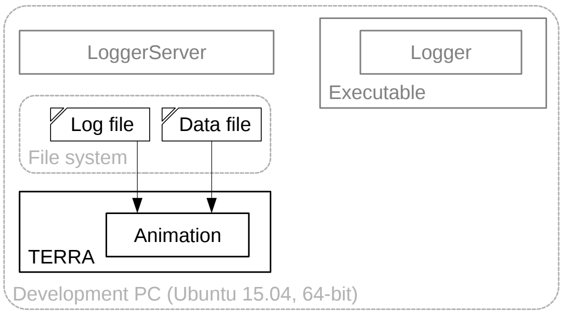
Due to time constraints, online animation has not been implemented. Therefore, the experiments are conducted as offline animation sessions, using the loggerserver to collect log data. For the purposes of these experiments, there is no difference between running the executable and / or the loggerserver on a development machine and running them on a target. The reason is that the logging facilities are pre-existing and known to provide the same log output independent of their exact distribution among machines. This is to be expected, as they communicate using TCP/IP regardless of the setup.

Given the above, it is most convenient to conduct the experiments on a single development machine. This setup is shown in Figure 3.1. The setup is similar to that presented Section 2.1, the difference being that the executable is run on the development PC (compare Figure 2.1). Again, active components relevant to animation are shown in black, other components in grey.

In Figure 3.1, the prefix 'real-time' has been omitted from the logger and executable to indicate that hard real-time execution is not supported on the development PC. This does not influence the order of execution; only timing is affected.



(a) Collection of execution flow data



(b) Animation using the collected execution flow data

Figure 3.1: Overview of the PC setup used to perform animation experiments

The test procedure is explained in Sections 3.1.1 to 3.1.3 and illustraded in the activity diagram shown in Figure 3.2

3.1.1 Obtaining animation data

The procedure is as follows. First, a model is created in TERRA. From this model, LUNA C++ code is generated, user code is added to code blocks insofar necessary, and an executable is built, using a LUNA build that includes real-time logging facilities. Next, the loggerserver is started. It waits for a real-time logger to connect. Once the executable has started, its logger connects to the loggerserver, which stores the log data in log files on the file system. Once the logs are complete, i.e. a sufficient number of iterations has been achieved, the executable and

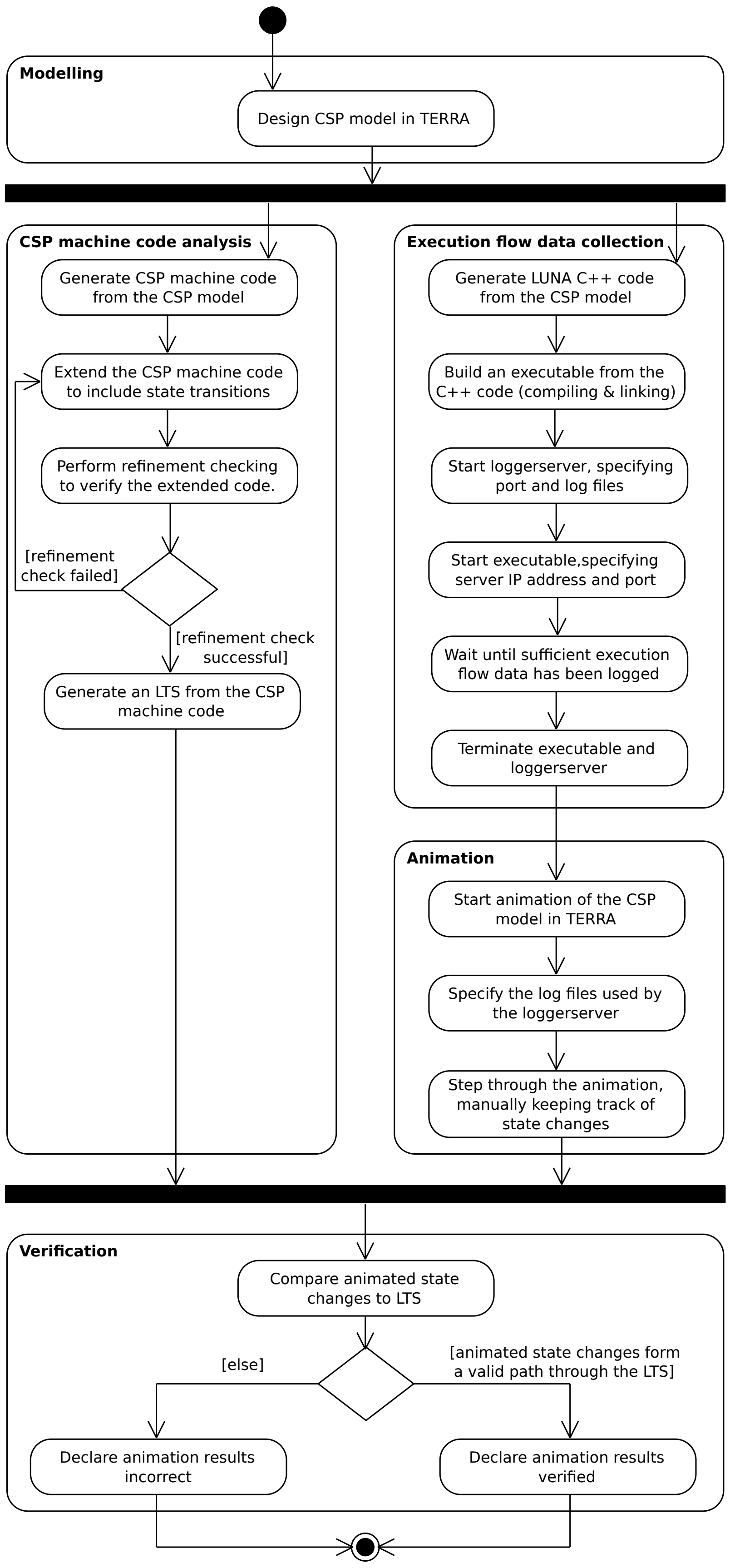
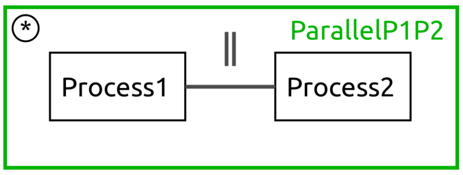

Modelling
Design CSP model in TERRA
CSP machine code analysis
Generate CSP machine code from the CSP model
Extend the CSP machine code to include state transitions
Perform refinement checking to verify the extended code.
[refinement check failed]
[refinement check successful]
Generate an LTS from the CSP machine code
Execution flow data collection
Generate LUNA C++ code from the CSP model
Build an executable from the C++ code (compiling & linking)
Start loggerserver, specifying port and log files
Start executable,specifying server IP address and port
Wait until sufficient execution flow data has been logged
Terminate executable and loggerserver
Animation
Start animation of the CSP model in TERRA
Specify the log files used by the loggerserver
Step through the animation, manually keeping track of state changes
Verification
Compare animated state changes to LTS
[else]
[animated state changes form a valid path through the LTS]
Declare animation results incorrect
Declare animation results verified

Figure 3.2: Activity diagram of the test procedure

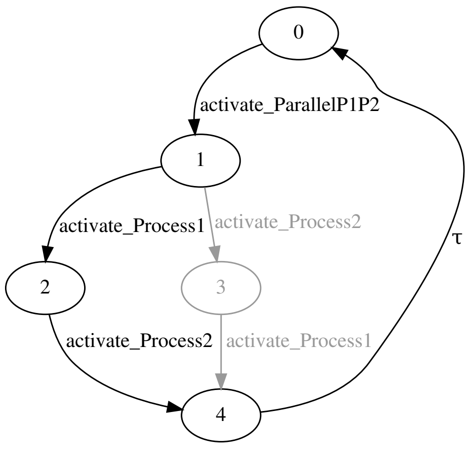
loggerserver are stopped. An animation is then started in TERRA, using the model and the log files.

3.1.2 Verifying animation data

To verify correctness of the animation results, CSP machine code is generated from the TERRA model, and analysed using FDR3. In FDR3, a Labelled Transition System (LTS) of the model can be generated. In an LTS, each node corresponds to one of the system's possible state configurations. Each edge is a named transition to another state configuration. See Figure 3.3b for an example.



(a) CSP model of Test case I



(b) LTS of Test case I

Figure 3.3: Test case I CSP model and LTS

The valid transitions from a state configuration are given in the LTS. Therefore, an LTS can be used to manually verify that the animation results are valid in terms of the CSP model.

LUNA's logging facilities define five types of states. For each individual element, the transition order of these states is fixed, although the exact order depends on the type of CSP construct. These states are *Activate*, *Run*, *Activating other processes*, *Waiting*, and *Done*. These states are discussed in Appendix A.

Due to this high number of states, combined with the randomness of parallel constructs, a model's state space quickly becomes too large to report on. Therefore, a simplified state space is used instead, as explained next.

If a model is deadlock-free, this means that each of its model elements can complete their iterations (unless the model element is not run at all, e.g. a non-activated Alternative branch). This implies that the corresponding states defined for that model element (as given in Appendix A) are visited. For example, if a Code Block finishes its iteration, this means that the states *Done* → *Activate* → *Running* → *Done* have been visited, in that order. Since this order is fixed, this can be condensed to two states, e.g.: *Done* → *Activate* → *Done*.

Given the above, the state space of a deadlock-free model can be condensed to a single state per model element, plus an (implicit) return to the model elements' initial configurations. Such a reduced state space is shown in Figure 3.3b.

There is a drawback to this approach. In reality, LUNA's scheduler can, for example, schedule a construct, A, to become active while another construct, B, is in its *Running* state. The reduced state space considers all of B's states (other than *Done*) to be the same state. Therefore, such scheduling subtleties are lost. This means that the condensed state space cannot be used to completely verify animation correctness. It does allow for verification of the order of activation, as well as deadlock freedom, in a manner that can be reported on.

The verification as presented here, therefore, is based on reduced state spaces and corresponding LTSes, supplemented with reporting on manual inspection of the execution flow animation.

In the reduced state space LTSes, a labeled transition to the condensed non-*Done* state of a process `P` is named `activate_P`. The reasoning is that it most closely resembles LUNA's *Activate* state, showing order of activation. The return to the *Done* state is not explicitly indicated.

To define these transitions, custom CSP code has been added to that generated by TERRA. Equivalence to the TERRA-generated code is proven using FDR3's refinement checking, as indicated in Figure 3.2.

3.1.3 Test coverage

Together, the models used in testing should contain all CSP constructs, TERRA's ports and TERRA's / LUNA's recursion construct and alternative guard types to achieve proper test coverage.

The model discussed in Section 3.3 achieves this. Since its state space is too large to show all relevant screenshots in this report, the experiment concept is illustrated using a smaller model, in Section 3.2.

3.2 Test case I

This model, shown in Figure 3.3a, consists of a parallel group of two processes, *Process1* and *Process2*. Due to a recursion on the parallel group, the model iterates indefinitely. Each iteration, either *Process1* or *Process2* is activated, after which the other process is activated. In the LTS of Figure 3.3b, this is shown by the two routes from state configuration `1` to `4`.

In the logged run of this model's executable, animation showed results consistent with the LTS. In the first iteration, *Process1* was shown to activate before *Process2*, corresponding to the leftmost branch in Figure 3.3b. This has been indicated by greying out the other branch.

Screenshots of the corresponding state configurations in the animations are shown in Figure 3.4. The figures' captions list the LTS configuration they correspond with. Since the recursion has been excluded from the LTS's edges, Figures 3.4a to 3.4c are all considered to be the initial configuration (labelled '`0`') in the LTS.

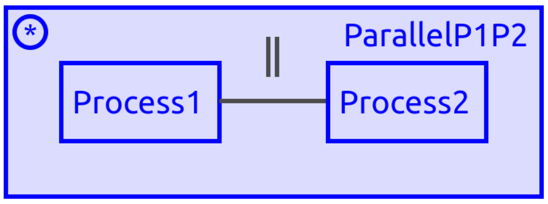
As shown in the figure, blue represents the *Done* state, green the *Activate* state. The additional turquoise state in between *Activate* and `Done`, hidden in the LTS, is `Activating other processes`. These colours are adjustable by the user.

Figure 3.4d shows that two processes can be running at the same time, which is, of course, legal in a CSP Parallel construct. The scheduler activates them one at a time, as can be seen from comparing figures Figures 3.4d to 3.4f.

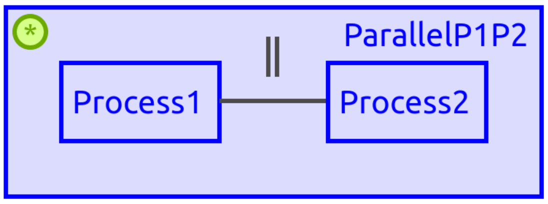
3.3 Test case II

Test case I has been used to illustrate the experiment procedure and to show that the animation behaves correctly for at least the subset of CSP constructs used in it. Test case II is more extensive. It contains all constructs except for a PriParallel. Additionally, Test case II uses Alternative constructs with expression-guards and channel-guards. The channel-guarded constructs are Sequential groups of which Readers are the first elements.

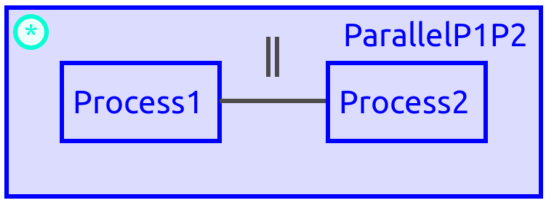
The model is shown in Figure 3.5. In *Producer*, a channel to write to is selected by the Code Block *CbChannelSelector*, which ensures that one of the three Writers' guard conditions evaluates to true. The Readers on the *AltConsumer* side are channel-guarded. The Alternative's branch selection in *Producer* therefore determines which of the consumer CodeBlocks is run in an iteration.



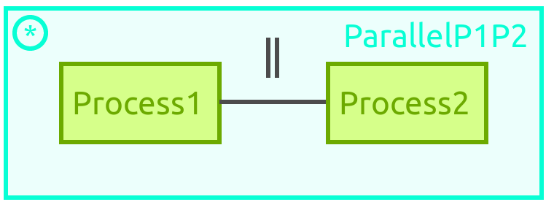
(a) Initial (*Done*) configuration (LTS node ‘0’).
Next configuration: Figure 3.4b
↓



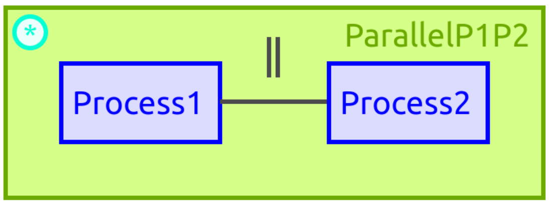
(b) Recursion *Activated* (LTS node ‘0’).
Next configuration: Figure 3.4c
↓



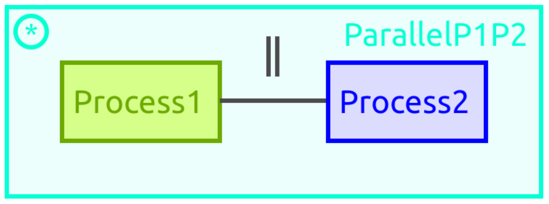
(c) Recursion *Activating other processes* (LTS node ‘0’). Next configuration: Figure 3.4e
↓



(d) `Process2` *Activated* (LTS node ‘4’).
Next configuration: Figure 3.4c
←



(e) `ParallelP1P2` *Activated* (LTS node ‘1’).
Next configuration: Figure 3.4f →

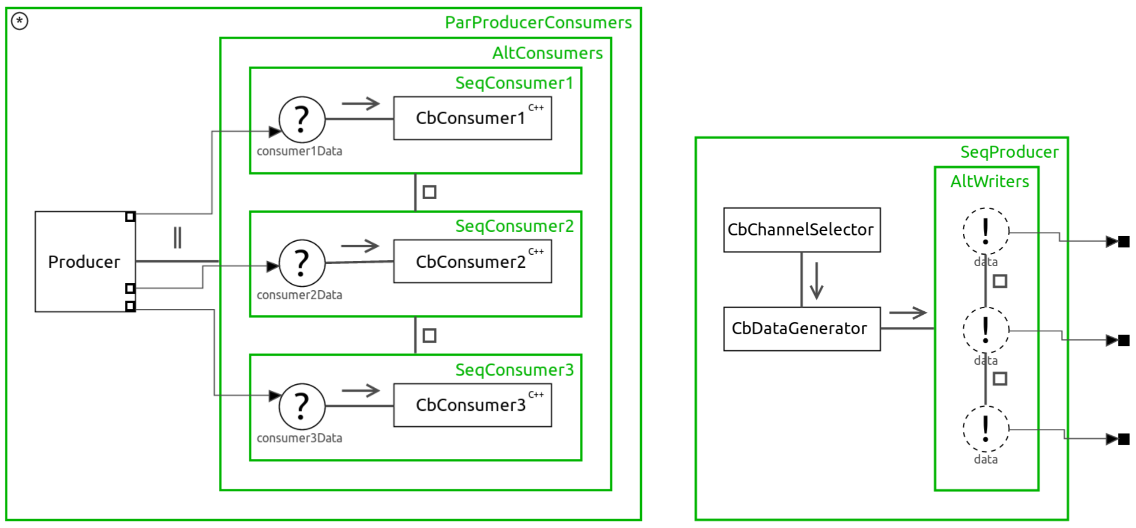


(f) `Process1` *Activated* (LTS node ‘2’).
Next configuration: Figure 3.4d ↑

Figure 3.4: Animation of Test case I state configurations. LTS node numbers refer to Figure 3.3b

The corresponding LTS is shown in Figure 3.6. The choice of channel and resulting Code Block execution is clearly visible from nodes 8–12. The branching options in nodes 1–8 and 13–16 are the result of the Parallel group *ParProducerConsumers*. The path taken in the first iteration, as observed using animation, is indicated by the black transitions and nodes. This (already reduced) state space is too large to include all corresponding figures. Instead, a single figure, showing the transition labelled ‘channel1’ is shown in Figure 3.7.

Figure 3.7b shows that *CbChannelSelector* and *CbDataGenerator* have completed their execution (blue), and *AltWriters* is activating (turquoise) its top Writer (green). On the top level, the Writers’ states are depicted by animation of the corresponding ports. The top level also indicates that *SeqConsumer1* has not been activated yet (i.e. it’s in the *Done* state from a potential previous iteration). The orange colouring indicates that those constructs are *Waiting* for their children to complete their execution.



(a) **(b)**

Figure 3.5: CSP model of Test case II. **(a)** Top level **(b)** Contents of the `Producer` process

3.4 Discussion

Animation of both Test case I and II showed transition sequences that are valid according to the models' LTSes, as indicated by the black paths in Figure 3.3b and Figure 3.6.

The additional state changes in-between those depicted in the LTSes have been manually determined to be correct.

During the experiments, it was observed that the state space is quite large, and that a lot of transitions need to occur per model element. This results in 'flooding' of the textual view, and distractions from the elements of interest in the graphical view. Therefore, it is desirable to provide settings to disable animation of certain states. For example, states *Running* and *Activating other processes* could be ignored, showing only *Activate*, and *Done*. *Waiting* is important for *Readers* and *Writers*, as it clarifies rendez-vous communication and is a good indication of deadlock issues. For all other constructs, *Waiting* could be disabled as well.

Another cause of confusion could be that the *Activate* state, in terms of implementation, means that the *Activate*d construct is placed in the LUNA scheduler's ready queue. Therefore, the scheduler is free to continue with another process first. It will often do so if a FIFO scheduling policy (with priority support) is used. FIFO is mandatory during animation, so this is always the case. While this is proper and intended scheduling behaviour, it makes execution flow harder to interpret by the user. In particular, focussing on a select set of processes becomes difficult, as context is often switched to other parts of the model.

In conclusion, there seems to be a trade-off between an accurate depiction of LUNA's CSP scheduling (which can be useful for debugging) and efficient, easily understandable animation.

A solution would be to group, or hide, states in TERRA, much like the reduced state space used in this chapter. For example, the *Waiting* state of compositional groups could be mapped to the *Activating other processes* state, as the distinction usually does not matter to the user: they already know that the compositional group will not reach the *Done* state until its children are *Done*. For Readers and Writers on the other hand, animating the *Waiting* state is essential, as it shows that no Writer or Reader is ready on the opposite channel end, which provides information on synchronisation and potential deadlocks.

Grouping multiple states should be implemented in TERRA and be optional. This way, LUNA log output provides the maximum amount of information, allowing the user to toggle state grouping on and off on the fly. This feature could be implemented in the database or the stepper and / or publisher.

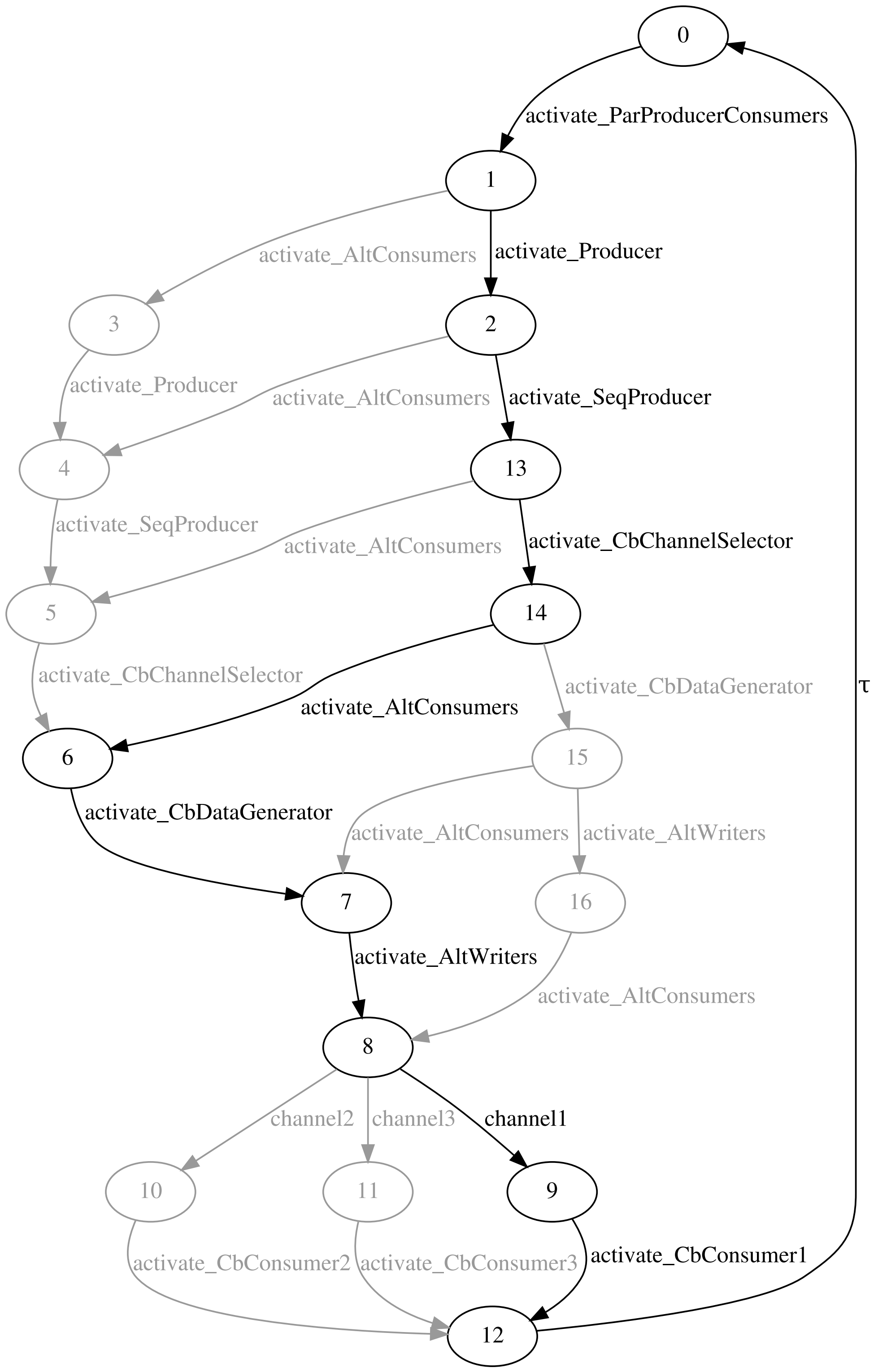
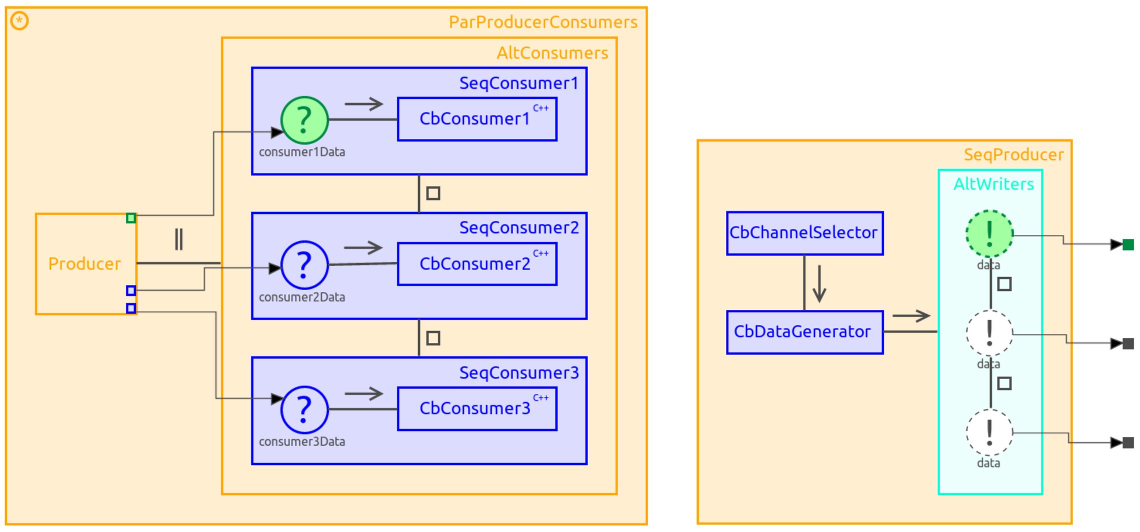


Figure 3.6: LTS of Test case II



(a) **(b)**

Figure 3.7: Animated TERRA CSP model of Test case II, in transition from configuration '8' to '9' (see Figure 3.6). **(a)** Top level **(b)** Contents of the `Producer` process

4 Conclusions and Recommendations

In this chapter, the thesis' results are evaluated, its added value is discussed and the extent to which the requirements have been realised is determined. Furthermore, recommendations are given for future work on the animation facilities.

4.1 Conclusions

The problem description given in Section 1.2 is revisited in Section 4.1.1, showing and evaluating this thesis' contributions. Secondly, the requirements established in Section 2.2 are evaluated.

4.1.1 Evaluation of thesis goal

Insight into the execution flow is essential for effective modelling. Without this insight, debugging is difficult and validation of the model is not feasible. Furthermore, experience with students shows that understanding CSP often proves difficult, a prime example being rendez-vous communication.

Using the animation facilities developed in this thesis, such insight has become more easily accessible for TERRA CSP models. The animation shows CSP processes' states in both a textual and graphical manner, allowing the user to inspect execution order and, for example, the state configuration when a deadlock occurs.

Synchronisation using rendez-vous communication is an essential part of CSP and can be difficult to grasp and analyse. As a result of the hierarchical views of TERRA, communication channels can cross diagram levels, potentially obscuring Readers and Writers from view, making analysis even more difficult. To compensate, animation of ports has been implemented. Animated ports mimic their channel end's Reader or Writer, such that both sides of the rendez-vous communication are visualised even when the connected Reader and / or Writer is not in view.

Experiences during testing show that the number of states is likely higher than the optimum value for general analysis and educational purposes. This could potentially limit the amount of insight gained through animation. A solution would be grouping multiple states together into a single state.

4.1.2 Requirement evaluation

In this section, the requirements established in Section 2.2 are evaluated.

General requirements

- Animation is possible, with support for stepping forwards and backwards, running, jumping, pausing and terminating. This satisfies Requirement 1. The animation facilities have been shown to work correctly (see Chapter 3).
- Offline animation (Requirement 2) is possible, using the pre-existing LUNA loggerserver to receive log data from the real-time logger and write it to disk. The resulting log files can be imported by TERRA and used for animation.
- Online ('live') animation has not been implemented, due to time constraints. Therefore, Requirement 3 has not been met.

Modelling suite requirements (TERRA)

- A graphical animation view has been implemented, satisfying Requirement 4. The graphical animation updates its view based on the user's stepping (or running or jumping), depicting all of LUNA's states by colouring and setting line thickness on the corresponding figures. Ports connected to a Reader or Writer on another diagram are animated according to that Reader or Writer's state, which eases analysis of CSP's rendez-vous communication.

- The textual animation view satisfies Requirement 5. It adds a view to the animation history and order of state changes, whereas the graphical view shows the state configuration at a certain point in time (i.e. a certain step number). Additionally, as it is a flat view, it allows the user to track state changes of model elements that are not visible on the graphical view's current diagram level.

- The user can change colours and line thickness for all states using Eclipse's preference menu. This largely satisfies Requirement 6. The textual view is not configurable. An important customisation feature that can be added is hiding of states (see Section 4.2).

Communication requirements (TERRA and LUNA)

- Direct communication between TERRA and the logger, Requirement 7 has not been implemented. This hinders usability, as the user is required to use an additional program and import the resulting log files in TERRA.

- Pre-existing tooling is used for logging, as direct communication between TERRA and the logger has not been implemented. The existing platform independence and limitations remain: there is platform independence in the sense that QNX, regular Linux and RTAI are supported by the real-time logging facilities. However, the real-time logger and the loggerserver must run on and be built using the same word size (e.g. both on a 64-bit platform). Thus, Requirement 8 is automatically partially satisfied, while no improvements have been made.

- No specific facilities for handling data loss (Requirement 9) have been implemented. If missing or illegal information is encountered in the logs, the user is notified and animation can continue up to the step before the incorrect information is encountered.

Non-functional requirements

- Per Requirement 10, using the animation facilities should be intuitive. Progressing through an animation is indeed intuitive and switching between animations is straightforward. However, the absence of online animation makes setting up an animation system cumbersome: it requires a separate command-line program to write two log files, which then have to be selected in TERRA for use in animation. Future additions can significantly improve intuitiveness. An example is using the textual view to select a step to jump to.

- The animation facilities comprise a set of Eclipse plugins that separate the base implementation and CSP-specifics. Furthermore, the design is such that additional views and visualisation facilities can be added in the future. Thus, Requirement 11 is satisfied.

- The animation facilities only use TERRA models to obtain information from, no models are changed. This satisfies Requirement 12.

4.2 Recommendations

This section provides recommendations for future work.

Firstly, direct communication between TERRA and the logger should be implemented, as described in Requirement 7. This is important for ease of use.

Secondly, several smaller improvements can significantly improve user experience. These improvements are listed below:

- *Wizard.* A wizard to start a new animation session would be more user-friendly than the current implementation, especially if it could remember previous settings.
- *Navigating via textual view.* Intuitively, one would expect that the textual view allows interacting with the animation. For example, jumping the animation to the selected step in the view, or opening the diagram that contains the selected step's state in the graphical viewer.
- *Indicating state changes on a different level.* In the graphical view, state changes often occur in a model element that is not visible in the current view. The view could, for example, indicate such state changes by decorating a parent or child process that is in view. This would guide the user, navigating them through the hierarchy to the process that changed state. Another option is adding a tree view of the model, which allows all model elements to be shown in a single view

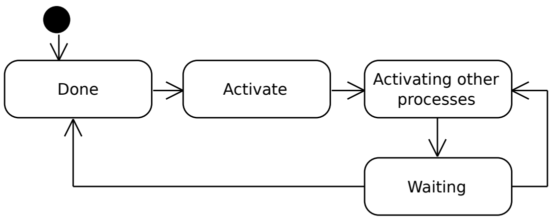
Support for architecture models and external models is essential for using the animation facilities in typical development scenarios. These scenarios usually include an architecture model connecting linkdrivers to an external CSP model, as well as control using 20-sim external models.

Logging and showing variable values can greatly increase debugging potential. This is especially true for variables that influence the execution flow, e.g. those used in guards and recursion conditions.

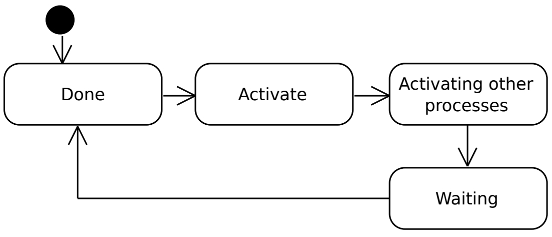
Lastly, the number of logged state changes is quite high, making orientation difficult and obscuring activation order. On the other hand, such a level of detail may be needed for debugging. Therefore, there should be options to include or exclude states from animations.

A LUNA CSP States

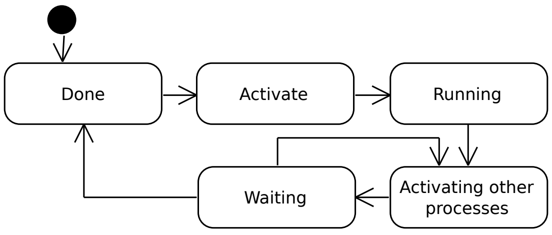
LUNA defines five possible states in total: *Activate, Running, Done, Waiting,* and *Activating other processes*. The possible states and transitions are different per type of CSP Construct, as shown in the state machine diagrams in Figure A.1.



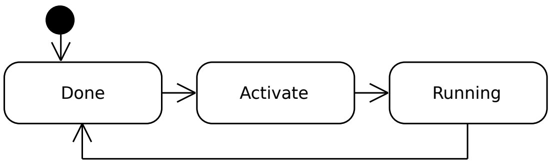
(a) State machine diagram of Sequential constructs



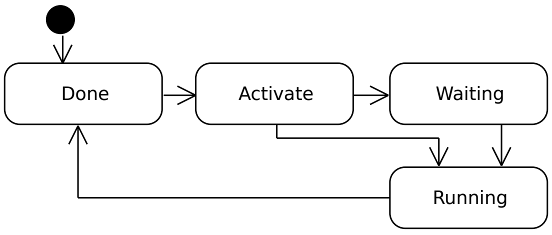
(b) State machine diagram of Alternative and Parallel constructs



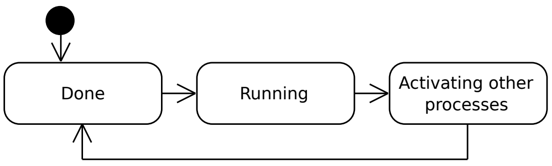
(c) State machine diagram of Recursions



(d) State machine diagram of Code Blocks



(e) State machine diagram of Readers and Writers



(f) State machine diagram of channel-guarded Readers and Writers guarding a Sequential construct

Figure A.1: State machine diagrams for the various CSP constructs, as defined in LUNA

The states have the following meaning:

- *Done* indicates that a CSP construct is inactive. This is the start and end state of each iteration of a CSP construct. Recursions are an exception (see Figure A.1c), as their iterations consist of transitioning between *Acting other processes* and *Waiting* (except for the first and last iteration).
- *Activate* is the activation of a CSP construct from its *Done* state.
- *Activating other processes* is mainly used for compositional groups, and refers to activating the CSP constructs contained in these groups, i.e. their 'children' (opposite: 'parent', which is always singular).

  As Sequential groups need to wait for a child to finish before activating the next, they transition between *Activating other processes* and *Waiting* until the last child has been activated (see Figure A.1a).

  Parallel constructs are free to activate all children. Therefore, they remain in state *Activating other processes* until all children have been activated. For Alternative constructs, *Activating other processes* applies to activating a single child, namely the (first) one of which the guard condition is met. Both Parallel and Alternative constructs consequently have the same state machine diagram, shown in Figure A.1b.

A channel-guarded Reader or Writer can be part of a Sequential group which is itself a child of an Alternative construct—hence the guarding. It must be the first construct in the Sequential group. In the LUNA C++ code, however, the Sequential group does not contain the channel-guarded Reader (or Writer). Rather, the Reader is the parent of the Sequential group. If the guard condition is met (i.e. a Writer is ready on the other channel end), the Reader and Writer perform rendez-vous communication. This happens in the *Running* state, as explained below. Next, the Reader activates the Sequential construct, thereby transitioning to the *Activating other processes* state. See Figure A.1f.

- *Running* applies to all non-compositional group constructs. For Readers and Writers, it signifies the occurrence of rendez-vous communication, and hence synchronisation and data transfer. For Code Blocks, it signifies execution of their code, e.g. 20-sim controller code.

- *Waiting* applies to compositional groups and Readers and Writers. For compositional groups, it indicates that the group is waiting for (one of) its children to finish execution, i.e. transition to the *Done* state.

  For Readers and Writers, it indicates that the Writer or Reader on the other end of the channel is not (yet) ready for communication. If a Reader or Writer is in its *Waiting* state, the opposite Writer or Reader can transition to *Running* without going through the *Waiting* state, as indicated in Figure A.1e.

Bibliography

Bezemer, M. M. (2013), *Cyber-Physical Systems Software Development - way of working and tool suite*, Ph.D. thesis, University of Twente, doi:10.3990/1.9789036518796.

Bezemer, M. M., R. J. W. Wilterdink and J. F. Broenink (2011), LUNA: Hard Real-Time, Multi-Threaded, CSP-Capable Execution Framework, in *Communicating Process Architectures 2011, Limmerick*, volume 68 of *Concurrent System Engineering Series*, Eds. P. Welch, A. T. Sampson, J. B. Pedersen, J. M. Kerridge, J. F. Broenink and F. R. M. Barnes, IOS Press BV, Amsterdam, pp. 157–175, ISBN 978-1-60750-773-4, ISSN 1383-7575, doi:10.3233/978-1-60750-774-1-157.
http://wotug.org/papers/CPA-2011/Bezemer11/Bezemer11.pdf

Bezemer, M. M., R. J. W. Wilterdink and J. F. Broenink (2012), Design and Use of CSP Meta-Model for Embedded Control Software Development, in *Communicating Process Architectures 2012*, volume 69 of *Concurrent System Engineering Series*, Eds. P. Welch, F. R. M. Barnes, K. Chalmers, J. B. Pedersen and A. T. Sampson, Open Channel Publishing, pp. 185–199, ISBN 978-0-9565409-5-9.
http://wotug.org/papers/CPA-2012/Bezemer12a/Bezemer12a.pdf

Clegg, D. and R. Barker (1994), *Case method fast-track: a RAD approach*, Addison-Wesley Longman Publishing Co., Inc.

Controllab Products (2015), 20-sim.
http://www.20sim.com/

Gibson-Robinson, T., P. Armstrong, A. Boulgakov and A. W. Roscoe (2014), FDR3 — A Modern Refinement Checker for CSP, in *Tools and Algorithms for the Construction and Analysis of Systems*, volume 8413 of *Lecture Notes in Computer Science*, Eds. E. Ábrahám and K. Havelund, Springer Berlin Heidelberg, pp. 187–201, ISBN 978-3-642-54861-1, doi:10.1007/978-3-642-54862-8_13.
http://dx.doi.org/10.1007/978-3-642-54862-8_13

Hoare, C. A. R. (1985), *Communicating Sequential Processes*, Prentice Hall International.
http://www.usingcsp.com/cspbook.pdf

IBM (2015), Rational Rhapsody Developer,
www.ibm.com/software/products/en/ratirhap, accessed: 2015-09-25.

Krasner, G. E. and S. T. Pope (1988), A Description of the Model-View-Controller User Interface Paradigm in the Smalltalk-80 System.

Noble, K. and L. Groves (1992), Tarraingím - a program animation environment, *New Zealand Journal of Computing*, **vol. 4**, pp. 29–40.

Rubel, D., J. Wren and E. Clayberg (2011), *The eclipse graphical editing framework (gef)*, Addison-Wesley Professional.

van der Steen, T. T. J. (2008), *Design of animation and debug facilities for gCSP*, Msc report 020ce2008, University of Twente.
http://essay.utwente.nl/58120/

Wilterdink, R. J. W. (2011), *Design of a hard real-time, multi-threaded and CSP-capable execution framework*, Msc thesis 009ce2011, University of Twente.
http://essay.utwente.nl/61066/