EINDHOVEN UNIVERSITY OF TECHNOLOGY

A communication protocol for interactively controlling software tools

Citation for published version (APA):
Wulp, van der, J. (2008). *A communication protocol for interactively controlling software tools*. (Computer science reports; Vol. 0823). Technische Universiteit Eindhoven.

Document status and date:
Published: 01/01/2008

Document Version:
Publisher's PDF, also known as Version of Record (includes final page, issue and volume numbers)

Please check the document version of this publication:

• A submitted manuscript is the version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher's website.
• The final author version and the galley proof are versions of the publication after peer review.
• The final published version features the final layout of the paper including the volume, issue and page numbers.

Link to publication

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain
• You may freely distribute the URL identifying the publication in the public portal.

If the publication is distributed under the terms of Article 25fa of the Dutch Copyright Act, indicated by the "Taverne" license above, please follow below link for the End User Agreement:

www.tue.nl/taverne

Take down policy
If you believe that this document breaches copyright please contact us at:

openaccess@tue.nl

providing details and we will investigate your claim.

Download date: 16. Nov. 2023

A Communication Protocol for Interactively Controlling Software Tools

J. van der Wulp

Technische Universiteit Eindhoven
PO Box 513, 5600MB Eindhoven, The Netherlands
j.v.d.wulp@tue.nl

Abstract

We present a protocol for interactively using software tools in a loosely coupled tool environment. Such an environment can assist the user in doing tasks that require the use of multiple tools. For example, it can invoke tools on certain input, set processing parameters, await task completion and have tools communicate the resulting output. It can also keep track of files produced by tools and prevent tools from reading and writing to the same file at the same time. The protocol serves as an interface between the tools and a central tool manager. Generally, the manager controls the tools and forms an interface to a human user. The protocol is used to connect our tool manager SQuADT to a variety of tools, hereby allowing these tools to be used on all major software platforms.

1 Introduction

The mCRL2 toolset (see [GMR$^+$07]) is a collection of tools around the formal modelling language mCRL2 that can be used for formal verification and analysis of process behaviour. Most of the tools have a traditional command line interface and today not everyone is comfortable with this way of working. Therefore we started working on a tool integration framework to make the toolset usable for a broader audience. The idea is that a uniform graphical user interface should make it easier to use tools without having too much knowledge about the specifics of every tool. The focus is simplifying the use of individual tools as well as combinations of tools and to automate frequently occurring tasks that involve the use of multiple tools.

The SQuADT desktop application is a graphical user interface layer around a new tool integration framework, within which a central part is played by the communication protocol described in this text. The name SQuADT, stands for Systems Quality, Analysis and Design Toolset, which refers to the kind of tasks that can be performed with the connected tools. The connected tools are those found in the mCRL2 toolset. Most of these tools can be used stand-alone by means of a traditional command line interface and some with a graphical user interface. The only communication between the tools is uni-directional by means of files or file streams (also known as piping). The design of the SQuADT graphical user interface as well as much of its functionality have, so far been targeted at tools with this specific behaviour.

The idea of using a graphical user interface to simplify the use of a toolset is is not new. The SQuADT application is very much influenced by the Eucalyptus application (see [JHA$^+$96]) in the CADP toolset. Eucalyptus was developed (around 1996) in the context of CADP for a very similar purpose as SQuADT is for the mCRL2 toolset. However, this does not mean that all the underlying ideas of SQuADT are the same as those of Eucalyptus.

Contrary to Eucalyptus, in SQuADT every action is performed in the context of a project. This approach is adapted from integrated development environments (IDE), see [Ecl, Net] for two popular examples. An IDE is an integration framework for software development; it integrates a number of often stand-alone software tools that are used for software development. In the project context SQuADT manages a collection of files and a collection of tools that can be used to add new files to a project. The user observes and directs this process through a graphical user interface. The core of that user interface is focused around an interactive visual overview of all data dependencies within a project. Every dependency represents an application of a tool on a set of files with another (disjoint) set of files as the result (where output depends on input). Within the context of a project SQuADT keeps track of tool applications and through it the dependencies between files in the project. The file dependencies are then used to monitor consistency (explained shortly).

Consistency is an relationship between input and output established by an arbitrary tool. Nothing is known about the exact relationship since nothing is known about the tool. So any change to either an input or output file can potentially violate consistency. Operations on files within a project can be monitored so operations that may violate consistency can be detected and signalled to the user automatically.

A file can be added to a project either by having the user select it as such or it can be produced by applying a tool to a set of files in the project. In the latter case the file is called *derived*. Ideally it should be possible to recreate all derived files from non-derived files (those added manually by the user). To this end we have imposed the restriction in SQuADT projects that tool applications may either modify or add files to a project, not both at the same time. When tool application results in creation of both new (output) files and modification of files in the project then all information regarding this tool application including the new output files is removed from the project. Basically this means that the project context can store all file dependencies in a project as a directed acyclic graph. Notice that we could have avoided the restriction for instance by adding a version attribute to every project file such that the dependencies between the versioned files form an acyclic graph. We decided against this approach because of the associated complexity and the fact that it does not add any immediate benefits for any of the tools in the mCRL2 toolset.

Monitoring consistency can help the user to find problems that result from the changes made to files in the project. This is not a novel idea. In much the same way IDEs monitor changes to files in a project in order to conservatively rebuild executables from source files. The main difference between the two is that an IDE takes care of generation and maintenance of all derived files, whereas in our case the user is expected to actively do these tasks.

Integrating tools works better 'when tools are aware' of the integration context. For example, tools and tool integration framework must cooperate in order to maximise the effectiveness of any inconsistency detection mechanism. Another example is an application built on top of an integration framework, that will likely offer a different user interface to the functionality provided by a tool (than for instance a command line interface). A different user interface may have different information requirements. For example in the context of a graphical user interface it is convenient and accepted practise to visualise state and progress. Showing state or progress when operating within the integration context may requires that a tool is tailored to this setting. Our communication interface is built on the idea of such a symbiosis between integration context and tool.

The following section introduces a number of important concepts and their connections. This is followed by a high level overview of the communication protocol. The purpose of this abstract perspective is to give the reader a picture of the structure of communication: what is communicated, in what way and why, without going into detail. Next is a more detailed description of the protocol, consisting of a description of the contents of messages, and their representation. At the end is a short comparison between integration frameworks that we know of (at least those with similar scope and purpose).

2 Concepts

It is only useful to consider integration between software tools when there is a meaningful way in which the tools can be used together. The purpose is then to obtain a result that cannot be obtained by any of the individual tools in isolation.

The tool integration problem can be characterised by the manner in which a given set of software tools can be used together in order to achieve a given goal. Notice that, when the goal is not compatible with the functionality (or any combination thereof) provided by the available tools, then the problem does not have a solution.

The SQuADT tool integration framework assists the user in solving tool integration problems. To this end the framework has functionality for execution of individual tools, monitors consistency of files produced as input/output of tool application, and it offers communication facilities for communication between a user and a tool. All of this functionality is tied to the communication protocol for interactively controlling software tools, which functions as interface between tool and tool integration applications such as SQuADT.

2.1 Tool

A *(software) tool* is a program that processes input and produces output that functionally depends on that input. Both input and output of a tool are sets of references to sources and sinks respectively consisting of binary data. The output of a tool is the result or accomplishment. In practise a data source/sink is often a file in the local filesystem, but it could also be a stream or data associated with user interaction through connected human interface devices. The latter are are treated specially.

Any tool is always used with a particular purpose in mind. Let's assume that the use serves the purpose (i.e. the tool is right for the job). A tool may serve different purposes and for each unique purpose the tool is said to have a *function* for that particular purpose. For all thinkable purposes, the largest set of functions for a particular tool makes up its *total functionality*.

2.2 Task

A *task* for a tool is the use of a specific combination of functions of that tool. This combination determines input and output requirements. The input of a tool needed for a task, called *task input*, is a non-empty set of resource identifiers (discussed shortly). Similarly output of a tool for a task, called *task output*, is a non-empty set of resource identifiers. Every input as well as output is associated with a type that is specified using the MIME format (Multipurpose Internet Mail Extensions, [Res01]). The input/output requirements for tasks typically include constraints on the types of inputs and outputs.

A resource identifier is a name of a file or stream associated. The principle method of specifying input/output files or streams is the Uniform Resource Identifier (or URI, see [MD02]). A URI that is specified as part of input must identify an existing resource before a tool can be applied. Similarly, a URI that is specified as output must identify an existing resource after the tool completes its task.

2.3 Task Configuration

The process of bringing a tool in the state where it can perform a specified task is called *task configuration*. After task configuration is complete the state of a tool can be made explicit by capturing it as a task specification. A *task specification* for a tool is a concrete specification that uniquely defines a task for that tool (without accounting for user-interaction).

A task specification, as depicted in figure 1, consists of a description of the task input/output and it specifies the specific combination of functions that define the task. The task input is a set of URIs that all identify an existing resource. The task output is a set of URIs of potentially non-existing resources. Every input as well as output is associated with a type. A tool may fail completing its task if the actual resource identified by the URI does not match the type.

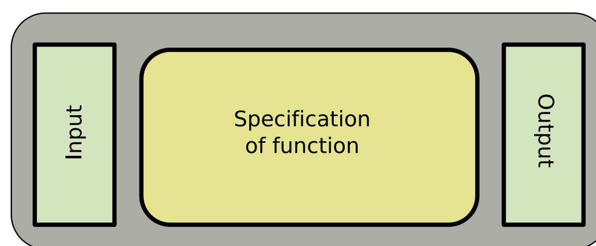


Figure 1: Graphic overview of the contents of a task specification

A tool creates its own task specification and communicates it afterwards with the integration framework. The framework can read and modify the part that specifies the task input and output, e.g. it can rename input files. The remainder of the task specification is tool-specific, the framework can only store this information but not interpret. The purpose of communicating the configuration with the framework is to have a means to preserve it.

Task execution is the process of using a tool to fulfil a configured task. To configure a task the tool must create a task specification in cooperation with the user and communicate it with the integration framework. To actually start execution the framework communicates a task specification with the tool that must either accept or reject it. Once a task specification is accepted actual task execution may commence.

2.4 Display

Since tools create task specifications themselves, with the user as beneficiary, the tool must have means of communicating with the user. The *(interaction) display* is a tool-controlled graphical user interface that acts as a direct communication channel between a running tool and the user. Every running instance of a tool has its own display. The display can also be used for instance to show task progress or to query a user during task execution.

3 Communication Protocol (high level)

The SQuADT application is built on top of a portable tool integration framework built in C++. At the heart of this framework is a communication protocol for interactively controlling tools. That is, SQuADT is responsible for controlling tools on behalf of a user; and the user can directly communicate with a tool through the display facility. So there are three communication parties: user, framework and tool.

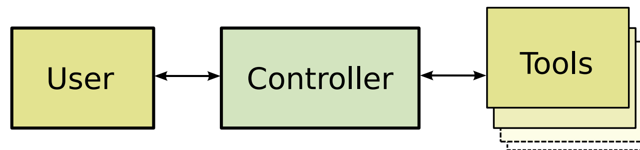


Figure 2: Communication parties

The protocol only concerns framework and tool, and from here on those parties are referred to as *controller* and *tool* respectively. The role of a tool is that of a configurable service, that of a controller is to orchestrate tool actions on behalf of the user.

The smallest unit of communication is a message. Messages are sent and received in a particular order and the protocol specifies how each message must be interpreted. Every message is equipped with a type that identifies its purpose. The type indicates how the message should be interpreted and provides a means to specify restrictions on message order. Interpretation of a message is based on both its type and the role of the party that receives it.

Message interpretation, besides type and role of the receiving party, is also affected by the messaging context. A *message context* is a chronologically ordered list of messages that were all either sent or received by the same communication partner in that particular order. More formally let p and q be communication partners. The *receiving context* of p (pertaining to q) is the sequence of *all* messages $m_1, m_2, m_3, \ldots$ that were sent by q and received by p in this particular order. The *sending context* of p (pertaining to q) is the sequence of *all* messages $m_1, m_2, m_3, \ldots$ that were sent by p and received by q in this particular order. From hereon we will assume that every message that is sent will also be received and that message order is preserved. More precisely, the receiving context of p pertaining to q is a prefix of the sending context of q pertaining to p.

A basic pattern used in the protocol is a request-response sequence. The tool, as well as the controller, can issue a request that the other party *must* respond to. So a request in the sending context can always be uniquely paired to a response in the receiving context. Besides this clear pattern there are also notification messages, e.g. messages that do not require any response.

The remainder of this section provides a high level overview of the protocol in terms of the different message types and their purpose. The next section zooms in on the concrete representation of the different messages.

3.1 Message Presentation

Messages have a name and are specified as a tuple consisting of a type, a direction and a specification of the structure of its contents. In practise, the type is meta-information that is kept on the message 'envelope' which hides the contents of the message. The direction is used for specifying which communication party is allowed to send such a message. In practise the direction is always clear for the party that sends/receives a message. Whether a message (contains data) can be established without 'opening the envelope' (or inspecting the data).

Message types are introduced on demand; they are names that serve to identify message purpose. The type determines the way in which a communication partner should interpret the data. The way in which data inside a message is structured is described in terms of named types that may be composed of other (nested) types. The type notation is adapted from the mCRL2-data syntax for specification of data types [GMR+07]. A BNF specification style is used to highlight the structure of the data and hint at the represented information. Concrete representational details are not discussed until section 4.

The names of the types in the data field hint at both the purpose and inter-dependencies between messages. There is a small number of standard types, namely `ID`, `URI`, `MIME-type`, $\mathbb{B}$, $\mathbb{N}$, and $\mathbb{S}$. An `ID` represents a textual identifier with the purpose of referring to communicated information objects across message boundaries. Types `URI`, `MIME-type` are string types that specify a URI [MD02] and MIME-type [FB96] respectively. Types may be composed using the $\times$ operator, as usual, and there are the parametrised types *Set* and *List* for specifying sets and lists (in the usual mathematical sense). Structured sorts are used for specification of sum types and compact representation of named product types.

The following two tables specify messages named 'example request' and 'example response' respectively. The request is sent by the controller and the response by a tool. The idea is that the tool computes distances for a trajectory specified by a list of time periods and relative orientation changes. Picture a tool that acts as a simulator for mars rover movement on a complex terrain.

example request	
message type:	`example`
direction:	tool to controller
data:	List(*struct* distances(minimum : $\mathbb{R}$, maximum : $\mathbb{R}$)

The request contains a list of pairs of `Duration` and `Direction`. Think of duration as time range type that coincides with the (actually positive) Real number domain. The direction is used for specification of relative orientation either forward for no change in orientation or some degrees left or right.

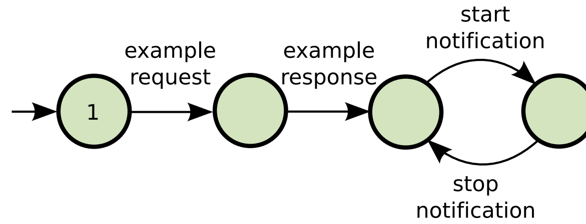
example response	
message type:	`example`
direction:	controller to tool
data:	List(Duration $\times$ Direction) Direction = *struct* left(Angle) $\mid$ straight $\mid$ right(Angle) Duration = $\mathbb{R}$ Angle = $\mathbb{N}$

The response consists of a list of pairs of minimum and maximum distances travelled. The result may of course depend on minimum/maximum acceleration initial trajectory velocity, and robot and terrain characteristics etc. The message content description in terms of types is simply used as high-level specification of input versus output of a tool for this specific task.

3.2 Communication behaviour

State diagrams are used to specify the allowed communication behaviour. An action label on a transition is the name of the message being sent. Special cases are connection initiation and severing transitions. An

incoming arrow from no state identifies the initial state. Figure 3.2 shows an example of how we specify allowed communication behaviour.



From the initial state an example request takes place, meaning that a communication action occurs between the partners. Subsequently an example response must occur, after which a state is reached from which a start notification can occur. A start notification can only be followed by a stop notification and vice-versa. The starting and stopping can be repeated ad infinitum.

3.3 Instance identification

Although the integration framework (controller-side) starts tools, this does not automatically mean that protocol communication with that tool has been established. The tool must initiate communication and the framework is to wait for this to happen. In the meanwhile the framework may be communicating with other tools. So when communication is established the controller must identify the its communication partner as a tool that was started previously. Instance identification is the means by which a tool (instance) must somehow identify itself to the controller.

Actual identification requires a pre-communicated secret. This secret takes the shape of an identification token, which is passed to the tool at startup. When the tool subsequently initiates communication, the first communication action consists of exchanging the identification token. The framework then validates the token and in case of failure it cuts off further communication.

A new messaging context (connection) is established for both partners when a tool initiates communication with a controller. The message is used for exchanging identification tokens; it looks as follows.

identification notification	
message type:	`identification`
direction:	tool to controller
data:	Token
	Token = $\mathbb{S}$

There is no response to such a message, after this message was communicated the controller-side has the initiative. The result is that either the connection is severed or the controller will send any command or request message (which will be treated shortly). Figure 3 shows the combined behaviour of a controller and a tool with regard to instance identification.

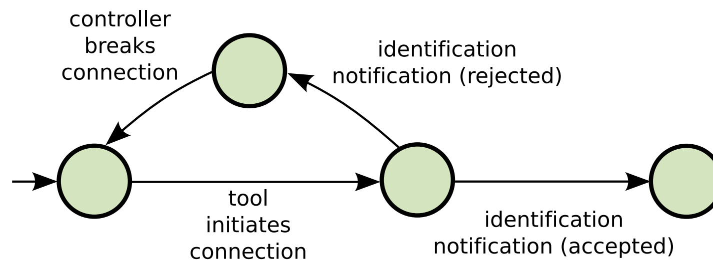


Figure 3: Process of instance identification

A tool initiates a connection, sends an identification notification and starts waiting for incoming messages. At the side of controller the value of the token is used to choose between breaking the connection and accepting the identity of the tool on the other side.

3.4 Capabilities

Capabilities represent both an information facility for one communication partner to learn about the capabilities of the other, as well as a protocol extension mechanism. The extension mechanism currently only consists of a means to check for the protocol version supported by a communication partner. This provides some limited backward and forward compatibility between protocol versions. For example it allows a tool developer to check controller side support for facilities that have been introduced in specific protocol versions. The exchange process follows the basic request-response pattern and is symmetric for both communication parties. A request for capabilities message looks as follows.

capabilities request	
message type:	`capabilities`
direction:	controller to tool
data:	

capabilities request	
message type:	`capabilities`
direction:	tool to controller
data:	

The partner that receives a capabilities request *must* send a capabilities response message. For a controller the response only contains the protocol version number and looks as follows.

capabilities response (controller)	
message type:	`capabilities`
direction:	controller to tool
data:	Version = *struct* version(major : $\mathbb{N}$, minor : $\mathbb{N}$)

The response for a tool additionally contains a sort of 'advertisement' of the tools functionality in the shape of a non-empty set of input configurations. An *input configuration* is a pair of a *category*, and a non-empty list of names for inputs associated with a type (storage format). The category is a descriptive name for the type of functionality that a tool offers for that specific input configuration.

capabilities response (tool)	
message type:	`capabilities`
direction:	tool to controller
data:	Version $\times$ Set(InputConfiguration) Version = *struct* version(major : $\mathbb{N}$, minor : $\mathbb{N}$) InputConfiguration = Category $\times$ List(`ID` $\times$ `MIME-type`) Category = $\mathbb{S}$

The exchange of capabilities follows a request-response sequence as illustrated by the following figure.

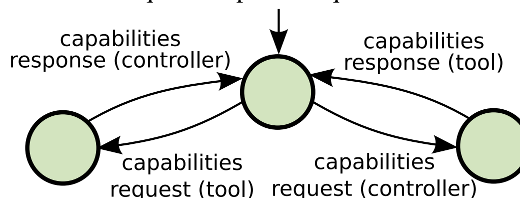


Figure 4: Process of exchange of capabilities

The set of input configurations partition the entirety of tasks that a tool can perform into classes that have the same input requirements and whose functionality falls in the same category. Every task specification is based on a single input combination. The input combination expresses a set of basic input requirements for configuration and abstractly characterises what functionality of the tool will be used.

3.5 Task Configuration

The process of task configuration starts as soon as a user selects a tool for use of some of its functionality. An input configuration (see section 3.4) serves as a concrete representation of the selected functionality and represents the starting point of further configuration. A task specification can be obtained in two ways, either by constructing it from an input configuration, or receiving one as the result of a task configuration process. Important to note is that every task configuration process starts by sending a task specification as part of a configuration request. The request message looks like:

configuration request	
message type:	`configuration`
direction	controller to tool
data:	Interactivity $\times$ Configuration Interactivity = $\mathbb{B}$ TaskSpecification = Category $\times$ List(ConfigurationItem) Category = $\mathbb{S}$ ConfigurationItem = *struct* object(`ID`,Object) $\mid$ option(List($\mathbb{S}$ $\times$ DataType)) Object = *struct* input(`URI` $\times$ `MIME-type`) $\mid$ output(`URI` $\times$ `MIME-type`) DataType = *struct* boolean $\mid$ string $\mid$ real_range($\mathbb{R}$,$\mathbb{R}$) $\mid$ integer_range($\mathbb{Z}$,$\mathbb{Z}$)

The request message consists of a pair of a Boolean, the *interactivity flag*, and a task specification. The interactivity flag specifies whether or not further configuration through interaction with the user is desired. Notice that a tool may initiate interaction with the user regardless the interactivity flag. The intended purpose however, is that when no interactivity is desired the user is consulted as little as possible. A more detailed explanation of a task specification follows after the introduction of the response message.

Configuration details, as part of task specifications, can now exist outside the tool. A consequence is that it opens up the opportunity that a tool receives an invalid task specification, e.g. it does not uniquely specify a configuration. Whatever the cause, it is necessary on the tool-side to check whether task specifications are usable. In other words a tool developer must provide a procedure to test task specification for validity. Moreover in the case a task specification is not valid the tool developer must resolve this problem through communication with the user.

Before sending a response the tool may initiate arbitrary interaction with the user (see subsection 3.9). The user as beneficiary is supposed to direct the process of task-configuration.

configuration response	
message type:	`configuration`
direction	tool to controller
data:	Validity $\times$ TaskSpecification Validity = $\mathbb{B}$ TaskSpecification = Category $\times$ List(ConfigurationItem) Category = $\mathbb{S}$ ConfigurationItem = *struct* object(`ID`,Object) $\mid$ option(List($\mathbb{S}$ $\times$ DataType)) Object = *struct* input(`URI` $\times$ `MIME-type`) $\mid$ output(`URI` $\times$ `MIME-type`) DataType = *struct* boolean $\mid$ string $\mid$ real_range($\mathbb{R}$,$\mathbb{R}$) $\mid$ integer_range($\mathbb{Z}$,$\mathbb{Z}$)

The response carries a judgement, the validity flag, and the final configuration. Depending on the value of the validity flag the embedded configuration was judged usable and the configuration is accepted. Notice that the task specification that is sent as part of a request is not necessarily the same as that in the response.

A task specification is modelled after the non-interactive part of a traditional command line interface. Traditionally non-interactive command line interfaces are used to capture the configuration of a program into a single string, the command. A command can be decomposed into an identifier of the program and a list of options that have a list of arguments. The purpose of a non-interactive command line interface is

exactly the same as that of a configuration specification. Namely capturing the configured state of a tool with the purpose of reproducing that state automatically at a later time.

A minimal task specification consists of a category specifier and a non-empty list of configuration items. The category specifier is a name that characterises the functionality of the tool, which is used to classify the tool in the user interface of an integration context. Configuration items are either objects (corresponding to input/output source, see figure 3.2) or options (as in their command line equivalents) with an arbitrary number of arguments. The options represent the language for task specification. A single option is a parametrised entity that represents the smallest part of optional configurable behaviour. The list of options identifies a combination of functions that make up the task.

The following figure depicts the basic (isolated) communication behaviour of the task configuration process.

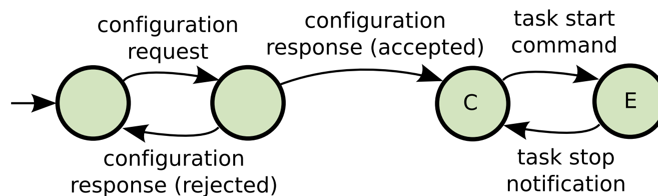


Figure 5: Schematic overview of the process of task configuration and execution

The process of task configuration is a straight-forward application of the request-response pattern. The state with label `C` represents the configured state, i.e. the state from which task execution may commence. Similarly the state with label `E` represents the state in which the tool is executing a task. Section 3.6 introduces the messages that deal with task execution.

3.6 Task Execution

When configuration is complete the controller may start task execution by sending a task start command. Configuration is complete when the controller receives a configuration response message with an accepted configuration and it has not sent a new configuration request. A task is called *in progress* as soon as a task start signal is sent, and as long as no task stop signal has been received. See figure 3.5 for a schematic overview of the process of task execution.

A message representing a task start command looks as in the leftmost table below. When a tool receives a task start command it must start executing the configured task. As task execution completes the tool must send a task stop notification as shown in the table on the right.

task start command	
message type:	`task`
direction	controller to tool
data:	

task stop notification	
message type:	`task`
direction	tool to controller
data	Result
	Result = $\mathbb{B}$

The data in the stop notification signifies success or failure of task execution. In case of failure the user should probably be notified of the details of the failure using the display or the reporting facility both of which will be discussed shortly.

3.7 Reporting

The purpose of the reporting facility is to inform the user (through the controller) of individual task activities and their progress. A report may be sent from any context and signifies either a warning, error or just notification of some event. The facility is intended as secondary source of information (next to the display) that a user may consult to get more feedback on configuration or task execution. A report message looks as follows:

report notification	
message type:	`report`
direction:	tool to controller
data:	ReportType × Description ReportType = *struct* notice | warning | error Description = $\mathbb{S}$

This facility is meant as an indirect method of communication with the user. The information from the reporting facility ends up in a log that is only visible when the user wants it. So the reporting facility must not be relied on as a part of the user regular user interface. The reporting facility is intended as additional source of information for the user and *not* an exception handling facility for the tool developer.

3.8 Termination

The termination facility allows the controller to terminate a tool in a controlled fashion. In this way a tool is allowed to free resources and remove inconsistent outputs. A termination command/request and the message that is sent as response are depicted below.

termination command	
message type:	`termination`
direction:	controller to tool

termination notification	
message type:	`termination`
direction:	tool to controller

The response such a request is a termination notification. The mandatory response signifies that the tool is shutting down (making preparations for termination) and will terminate soon. Additionally the notification can also be send by a tool when it is shutting down for other reasons than after a prior termination request. This functionality is meant to be used only in exceptional cases such as that a tool must terminate after an unrecoverable error.

The following figure shows the communication behaviour with regard to termination commands and notifications.

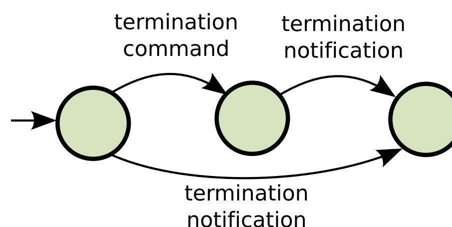


Figure 6: termination behaviour

When a tool fails to respond to a termination request, the integration framework may force a tool to terminate by other means.

3.9 Display

The display facility represents the primary means of a tool to communicate with the user. Think of it as an interactive bulletin board containing a set of user interface primitives (called *widgets*) in some arrangement. Every change to the widgets as a result of user interaction is directly communicated with the tool. The set of widgets on the display and their arrangement are controlled by the tool that owns the display. A *layout specification* is a description of a set of widgets and a set of constraints of how to position them relative to each other on the space made available by the display. A message that looks as follows is used to communicate layout specifications.

display change command	
message type:	`display_layout`
direction:	tool to controller
data:	LayoutManager LayoutManager = BoxLayoutManager BoxLayoutManager = *struct* horizontal(ElementList) | vertical(ElementList) ElementList = List(LayoutConstraints $\times$ `ID` $\times$ LayoutElement)) LayoutConstraints = Visibility $\times$ Status $\times$ Margins $\times$ Alignment Visibility = *struct* visible | hidden Status = *struct* enabled | disabled Margins = *struct* margins(top : $\mathbb{N}$, left : $\mathbb{N}$, bottom : $\mathbb{N}$, right : $\mathbb{N}$) Alignment = HorizontalAlignment $\times$ VerticalAlignment HorizontalAlignment = *struct* left | centre | right VerticalAlignment = *struct* bottom | middle | top LayoutElement = *struct* layout_manager(LayoutManager) | widget(Widget) Widget = *struct* progress_bar($\mathbb{N} \times \mathbb{N} \times \mathbb{N}$) | radio_button($\mathbb{S} \times \mathbb{B}$) | button($\mathbb{S} \times \mathbb{B}$) | checkbox($\mathbb{S} \times \mathbb{B}$) | label($\mathbb{S}$) | text_field($\mathbb{S}$)

Every widget *must* have an identifier, that uniquely identifies it in a layout. Subsequent communication of changes to the state of a widget rely on the identifier as widget specifier. A special subsection 3.9 is devoted to explaining the structure of a layout specification. For now we focus on communication of display interaction data and changes to the state of widgets on the display.

User interaction with widgets on the display is relayed to the associated tool as soon after the interaction took place. On the other hand a tool can change the internal state of widgets, e.g. change the label of a button from 'okay' to 'cancel'. In both cases information from individual widgets, called *display data*, is exchanged between the communication partners. A tool can request a state change for a set of widgets on the display using a message that looks as follows.

display manipulation command	
message type:	`display_data`
direction:	tool to controller
data:	List(`ID` $\times$ Widget) Widget = *struct* progress_bar($\mathbb{N} \times \mathbb{N} \times \mathbb{N}$) | radio_button($\mathbb{S} \times \mathbb{B}$) | button($\mathbb{S} \times \mathbb{B}$) | checkbox($\mathbb{S} \times \mathbb{B}$) | label($\mathbb{S}$) | text_field($\mathbb{S}$)

When a controller receives such a message it is interpreted as a state change of a widget that matches the identifier. The controller manages the display on behalf of a tool and needs to process these updates as follows. Let (id, s) be a pair of identifier and widget state specification. If id does not identify a widget on the display the state update represented by the pair is ignored. If id identifies a widget on the display and the type of this widget is not the same as that of s the state update is ignored. Otherwise s becomes the new state of the widget on the display that is identified by id.

As noted before, changes to widgets on the display as a result of user interaction are directly communicated with the tool. The tool does not have direct access to the display and is assumed to a local representation of the contents of the display in order to interpret the results. Communication of changes due to interaction is performed with a message that looks like:

display interaction notification	
message type:	`display_data`
direction:	controller to tool
data:	List(`ID` $\times$ Widget) Widget = *struct* progress_bar($\mathbb{N} \times \mathbb{N} \times \mathbb{N}$) | radio_button($\mathbb{S} \times \mathbb{B}$) | button($\mathbb{S} \times \mathbb{B}$) | checkbox($\mathbb{S} \times \mathbb{B}$) | label($\mathbb{S}$) | text_field($\mathbb{S}$)

When a tool receives a display interaction notification it is interpreted as a state change of the widgets that matches any of the identifiers. Let (id, s) be a pair of identifier and widget state specification. If id does not identify a widget on the display the state update represented by the pair is ignored. If id identifies a widget on the display and the type of this widget is not the same as that of s the state update is ignored. Otherwise s becomes the new state of the widget on the display that is identified by id.

Initially the display contains no widgets. User interaction without widgets is not possible, so display interaction notification messages will not be sent by the controller. Similarly display manipulation requests will not be sent by a tool and otherwise will be ignored. The figure below shows the communication behaviour regarding use of the display facility.

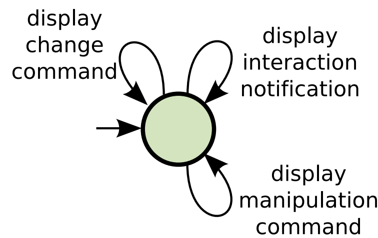


Figure 7: Communication behaviour with regard to use of the display facility

A display manipulation command is ignored when the controller cannot identify the widget that was targeted or when the state is not a valid state description for the targeted widget. The display becomes non-empty when a layout change command with a valid non-empty layout specification is communicated. A display interaction notification is ignored when the tool cannot identify the widget that was targeted or when the state is not a valid state description for the targeted widget.

Widgets and Layouts

The display facility shows an arrangement of widgets with which the user can interact, e.g. a button. To keep the message structure simple only a small set of basic widgets is supported. A very basic relative positioning scheme is available for positioning widgets in the available display space. As we have seen, altering the contents of the display is only possible by substituting one set of widgets and a layout for another. Furthermore the state of widgets can be altered by a tool, their arrangement (their layout) is immutable.

A layout specification subdivides the available space by recursively assigning space to so-called layout managers. A *layout manager* specifies the way in which elements are laid out across the display. The display always contains at least one layout manager, called the *top layout manager* that indirectly contains all other elements on the display. The top layout manager partitions the available space of the display to

its child elements. The elements of a layout manager consist of either widgets or layout managers that themselves may contain a number of elements.

The following figure shows an example of how widgets can be laid out using nested layout managers.

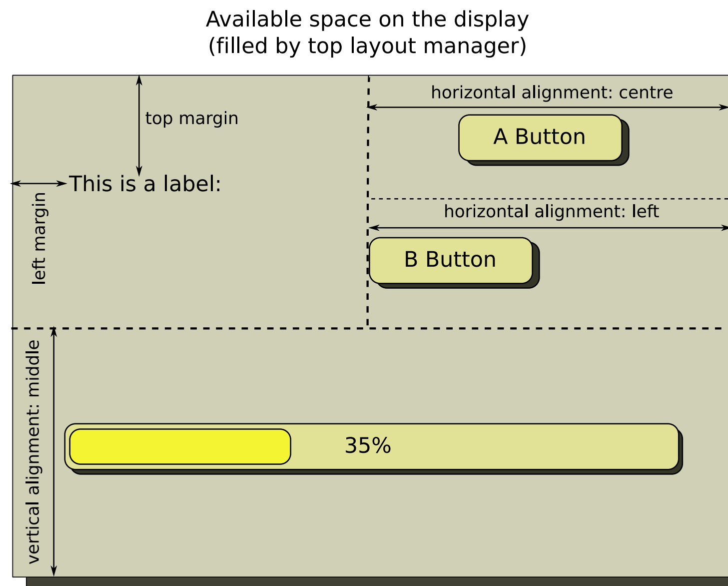


Figure 8: Example layout with annotation

The dotted lines in figure 8 mark the boundaries of space allocated to different layout managers at the same nesting level. A *box layout manager* is a special type of layout manager that arranges its child elements on the screen either horizontally, or vertically and expands elements (to fill space) in the direction perpendicular to the chosen direction.

Besides the layout managers that distribute available space among widgets there are also layout properties for further control over how elements are positioned and whether they are visible/hidden and usable (mutable/immutable for the end user). Figure 8 also illustrates the use of some of the layout properties. Element visibility determines whether this element is visible or not. The effect can can be used to create empty spaces with the dimensions of the invisible widgets. A widget can be enabled or disabled for user interaction. A widget is called *active* or *enabled* when the user can interact with it, otherwise it is *disabled*. Actual positioning properties: alignment, margins (in pixels) one of (top, right, bottom, left), vertical alignment (top, middle, bottom), horizontal-alignment (left, center, right).

3.10 Protocol Extension

Protocol extension is supported in the form of adding new message types and changing the structure for any of the existing message types. Depending on the nature of the changes it is necessary to increase the major or minor component of the protocol version tag (as returned after a capabilities request). Backwards incompatible changes require an increase of the major component. Naturally the changes to the version number are supposed to make it easy to test for additional functionality and/or to implement a fall-back mode for compatibility. Extension to the information exchanged with the capabilities facility can be used to provide means and check for even finer degrees of compatibility.

4 Implementation details

Up until here we have presented only a high level view of the communication protocol for controlling tools in our integration framework. This section focuses on the implementation details and addresses important design decisions.

Our communication protocol represents an interface between a our rather abstract notion of a tool and a tool integration framework. An important design goal was to make the use of this interface as simple as possible e.g. to not restrict its use to specific operating systems or programming languages. For implementation we have only looked at established inter-process communication mechanisms available as part of standard facilities provided by operating systems such as sockets and pipes. To further simplify the use of the interface from other programming languages it was decided to create a text-based message format based on XML [YPB$^+$06].

We decided not to pay to much attention to security aspects. At the time we felt that putting a focus on security would have slowed down the development process too much. The main goal has been on getting a proof-of-concept implementation of a communication interface on top of which to build the SQuADT application.

The OSI model [DZ83] is a popular way of analysing and describing communication protocols. Our use of this model only serves to provide a frame of reference. The OSI model divides communication into subproblems (using seven layers) that can be solved independently. Layers 1 through 5 represent basic functionality covered by widely available standard communication protocols. Those five layers provide a data communication connection between applications along which data can be transported. Our communication protocol covers the 6th (presentation) layer. The integration framework that is used as part of the SQuADT application covers the 7th (application) layer.

4.1 Transport

For transport of data layers 1 through 5 of the OSI model the Internet (or TCP/IP) protocol suite is used (see [Pos81b] and [Pos81a] for TCP(-v4) respectively IP). The TCP/IP protocol provides reliable bi-directional order-preserving delivery of a byte stream (layers 1 through 4 of OSI). TCP/IP also offers session functionality (5th layer OSI) or a connection between applications on top of which a messaging context can be implemented. As a result one-time tool identification (section 3.3) is possible, i.e. when a new messaging context is established.

4.2 Messaging

The presentation layer of the OSI model is about mapping between application level concepts (with their own syntax and semantics) and data representation in communication (data in messages). The topic of this subsection is the representation of the messages presented in section 3 and their interpretation in the domain of the application.

4.2.1 Basic Structure: Envelope

Messages are wrapped in the *message* element. A mandatory attribute is *type* that specifies the type of the message. The type attribute can occur only once and its value must be among those introduced in the previous section: *identification*, *capabilities*, *configuration*, *display_layout*, *display_data*, *termination*, *task*, *report*. As an example consider the following message with type 'termination'.

```
<message type="termination"><![CDATA[message content]]></message>
```

The content of a message is wrapped in a so-called CDATA section. The contents of a CDATA section is treated as character-only data and not parsed as markup. This allows embedding arbitrary character data into XML documents. To deal with data that contains fragments that match the end-marker `]]>`, any instance of `]]>` in the message content *must* be replaced by `]]]><![CDATA[]>`.

Below the message structure is specified for all the message types. Usually the structure of an XML document is specified using XML Document Type Definition ([CFT+06]) or the XML Schema standard [ACDP06]. An XML Schema specification for the complete set of messages is available but not included. For presentation purposes either of the two specification methods is very suboptimal; so we use tables to introduce the different elements and their usage constraints.

4.2.2 Capabilities

Capabilities are exchanged to inform each of the communication partners about the precise capabilities of the other. A request for capabilities is an empty message of type *capabilities* that looks as follows.

```
<message type="capabilities"></message>
```

Depending on which party sent the request message the response message looks different but it carries at least the protocol version. A response message as sent by the controller looks as follows:

```
<message type="capabilities">
 <capabilities>
  <protocol-version major="1" minor="0" />
 </capabilities>
</message>
```

In subsequent examples the message tags will often be omitted if the type of the message is clear from the context. The table below describes the protocol-version element. All attributes are mandatory except for those that are marked with *.

element: protocol-version	
attribute	*description*
major	integer that represents the major version component
minor	integer that represents the minor version component
contents: empty	

Table 1: XML element descriptions for protocol version

Any other child elements of the capabilities element must be ignored. The response as sent by a tool looks differently. The capabilities element must contain a non-empty set of input-configurations. An input-configuration is represented by means of the *input-configuration* element.

element: input-configuration	
attribute	*description*
category	short string that specifies a category
contents: an arbitrary number of object elements	

element: object	
attribute	*description*
id	string without white-space
format	MIME-type that specifies a storage format
contents: empty	

Table 2: XML element descriptions for input configurations

As an example consider the following fragment that represents the contents of a capabilities response message. As input the tool can take a file in the text based format called ‘mcrl2’ and then behaves either as an editor or as a visualiser. Alternatively it can also take a file in the binary ‘lps’ format and behave as an editor.

```
<capabilities>
 <protocol-version major="1" minor="0" />
 <input-configuration category="editing">
  <object id="mcrl2_in" format="text/mcrl2" />
 </input-configuration>
 <input-configuration category="visualisation">
  <object id="mcrl2_in" format="text/mcrl2" />
 </input-configuration>
 <input-configuration category="editing">
  <object id="lps_in" format="application/lps" />
 </input-configuration>
</capabilities>
```

4.2.3 Configuration

By design, our integration framework is oblivious to the format of any data (as in files) that are produced by connected tools. As noted earlier, the reason for this is generality in order not to inhibit applicability of the framework for software tools developed elsewhere. There is a critical dependency from framework on tool to supply complete information on files that have been produced as output and the format of these files. The configuration process and the resulting task specifications play a critical part in this.

Task specifications are both the source for dependencies for running tools in a project as well as for information about file formats. Initially there is no knowledge about any specific data format. MIME-type specifiers contained in task specifications provide the necessary information on the data format used in output files. The primary motivation for adopting the use of the MIME standard for representing the type of data sources is that it contains additional information about what the data represents. When used properly it can be used to determine whether the data represents a video stream or whether it is stored as text. In addition it gives opportunities for interoperability with other software applications such as the use of text editors or web-browsers.

Before looking in detail to the configuration related messages we have a quick look at typed arguments to options. The purpose of adding types to options is to allow automated sanity checks for checking task specifications. When an argument is of type Boolean but it was supposed to be an integer this mismatch can be established automatically. The details of how the available types are represented are shown below.

element: boolean
contents: either empty and otherwise ‘true’ or ‘false’

element: string
contents: an arbitrary string

Table 3: XML element description: boolean, string

An empty element e.g. `<boolean />` is a place holder for a value of the appropriate type. Concrete examples for a Boolean argument are `<boolean>false</boolean>` and `<boolean>true</boolean>`. Since all values are represented as strings these first two types are not very interesting by themselves. Numeric arguments are more interesting and expected to occur quite commonly.

element: integer-range	
attribute	*description*
minimum	an integer number in decimal notation
maximum	an integer number in decimal notation
contents: either empty	

element: real-range	
attribute	*description*
minimum	a real number in decimal notation
maximum	a real number in decimal notation
contents: either empty	

Table 4: XML element description: integer-range, real-range

Any tool will probably use the finite approximations of the numeric types that are supported in hardware. The two supported range types represent intervals over the integers and real numbers.
A configuration request consists of a single configuration section from which a task specification can be obtained. A configuration section consists of configuration element which has the following properties.

element: configuration	
attribute	*description*
interactive*	Boolean specifying whether interactive (re)configuration is desired
valid*	Boolean specifying whether the contents is a valid task specification
category	short string that specifies a category
contents: arbitrary number of option or object elements	

element: object	
attribute	*description*
id	string without white-space
type	either ‘input’ or ‘output’
location	URI that specifies a file in the local filesystem
format	MIME-type that specifies a storage format
contents: empty	

element: option	
attribute	*description*
id	string without white-space
contents: any sequence of boolean, integer_range, real_range or string	

Table 5: XML element description: configuration, object and option

The object and option elements correspond to sources of input/output and options respectively. The sequence of arguments that can be specified as children of *option* represent the list of arguments to this option.

The contents of a response message is exactly the same as that of the request, with the exception that attribute *valid* is mandatory in a response message. The following fragment shows an example configuration request, recognisable by the absence of the *valid* attribute.

```
<message type="configuration">
 <configuration interactive="true" category="debugging">
  <option id="-v">
   <integer_range>1</integer_range>
  </option>
  <object id="in" type="input" location="/dev/random" format="application/octet-stream"/>
  <object id="out" type="output" location="/tmp/out" format="application/octet-stream"/>
 </configuration>
</message>
```

A valid (initial) configuration can be obtained from an input configuration as follows. Create an empty *configuration* section and add an attribute *interactive* set to true and add the contents of an input-configuration section (section 4.2.3).

4.2.4 Display

Display manipulation is restricted to replacing the entire content of the display at once or modifying the state of individual widgets on the display. In particular it is not possible to manipulate the layout itself. More complete manipulation capabilities rapidly increase complexity. A conservative approach was chosen to keep initial complexity low and save on development time.

A small set of graphical user interface components (widgets) is available for tool developers to choose from for constructing display layouts. The downside of this limited choice is that a tool developer has little choice for constructing a graphical user interface using the display facility. Nevertheless layout construction and manipulation are by far the most complex functionality the protocol currently has to offer. Should the need arise, adding new widgets should be easy, however the protocol will have to be refined in order to realise this.

Widgets

We assume that the reader is familiar with the purpose and basic functions of each of the widgets presented earlier. Nowadays the use of graphical user interfaces is ubiquitous. The name and function of widgets we use and the way in which widgets are put together in a layout are based on concepts and terminology used in Java Swing.

Every widget has a mandatory *id* attribute that must be unique within the scope of the containing *display-layout* section (introduced shortly). The following two tables specify the XML elements that correspond with a label and button.

element: label	
attribute	*description*
id	mandatory identifier
contents: text-only	
The text for the label.	

element: button	
attribute	*description*
id	mandatory identifier
contents: text-only	
The text on the button.	

Table 6: XML element descriptions for label (left) and button (right)

As an example consider the following XML fragment that describes a label, a button and a checkbox (see table 4.2.4) all with with text “Cancel”.

```
<label id="x"><![CDATA[Cancel]]></label>
<button id="y"><![CDATA[Cancel]]></button>
<checkbox id="z" checked="true"><![CDATA[Cancel]]></checkbox>
```

Notice how the id attribute is unique for all the elements. This to be the case for all widget elements in every layout specification. When a button is pressed, or a checkbox is toggled this fact is communicated by sending a `display_data` message. The contents of this message is the complete widget specification.

A more interesting widget is the checkbox. The checkbox has a description and is always in one of two states: checked or not.

element: checkbox	
attribute	*description*
id	mandatory identifier
checked	optional Boolean argument for presence of tick mark
contents: text-only	
The text for a label that accompanies the checkbox.	

Table 7: XML element description for checkbox

The radio button widget is a more complex entity, because it is not a stand-alone widget. Radio buttons are always grouped and only a single button in the group is selected (pressed). By default the radio button in a group is selected that is found the highest (textually) in a layout specification. A radio button element is specified as follows.

element: radio-button	
attribute	*description*
id	mandatory identifier
connected	matches the **id** attribute of another radio button element
selected	optional Boolean attribute that represents whether the radio button is checked
contents: text-only	
The text for a label that accompanies the radio button.	

Table 8: XML element description for radio-button

The *select* attribute can be used to select a different button. A radio button group is formed by means of the connected attributes in all of the radio buttons in the group. Every *connected* attribute identifies another radio button in the group, within a group the *id* of every radio button occurs exactly once as value of a *connected* attribute. Every radio button in the group can be found by repeatedly following the *connected* attribute to find the connected radio-button by its identifier. If the selection changes then only the specification of the radio button that gets selected must be sent by means of a `display_data` message to inform the other side of this event. Such an update message could look as follows:

```
<radio-button id="y" connected="x" selected="true"><![CDATA[second]]></radio-button>
```

A text field displays an input widget for the user to input text. A specification of the text-field element:

element: text-field	
attribute	*description*
id	mandatory identifier
contents: text-only	
The initial text for the text field.	

Table 9: XML element description for text-field

The progress bar is used to show progress to a user. It models progress by means of a sub range of the integer domain, specified by a minimum and maximum value and shows progress by colouring part of this domain up to some 'current' value that *must* be in the domain [*minimum*...*maximum*]. The element looks as follows

element: progress-bar	
attribute	*description*
id	identifier
minimum	the minimum, integer value
maximum	the maximum, integer value
current	current state of progress as an integer value in range [minimum,maximum]
contents: empty	

Table 10: XML element description for progress-bar

Updates to the state of a widget are specified in the same way as in the layout specification. The *id* attribute identifies the widget of which the state is to be updated. The attributes then specify the new value for the attribute with the same name and child elements specify other aspects of the state. When attributes are missing, their value remains unchanged.

Layout

A display layout specification is represented by a *display-layout* element that contains a single *layout-manager* element, the top layout manager.

```
<display-layout>
 <layout-manager>
  <box-layout-manager variant="vertical" id="x">
   ...
  </box-layout-manager>
 </layout-manager>
</display-layout>
```

The *box-layout-manager* has a *variant* attribute that specifies the direction in which the elements directly contained in it are laid out on the available space. Every child element is associated with a value for each of the available layout properties. An implicit set of default values is assumed that can be used to reduce specification size. The default properties are as follows: alignment is left, no margins, elements are enabled and visible. The effective properties of an element are relative to that of the previous child. For example:

```
<box-layout-manager variant="vertical" id="">
 <properties margin-top="1" margin-bottom="1" horizontal-alignment="right" />
 <button><![CDATA[Ok]]></button>
 <properties />
 <button><![CDATA[Cancel]]></button>
</box-layout-manager>
```

The layout properties for both buttons are the same, top and bottom margins are one pixel, vertical alignment is middle and horizontal alignment is right and both elements are visible and enabled. The *properties* element directly preceding a widget specifies the layout constraints/properties for that widget. When there is no properties element (or it is empty) then the last specified values for each of the properties are in effect.

As a non contrived example there is a complete listing of the XML specification that can be used to generate a layout similar to the one depicted in figure 8 is given in appendix A.3.

5 Comparison

How does our approach measure up to other approaches to tool integration? In Eucalyptus, the graphical front-end to CADP, detailed knowledge about the capabilities of individual tools as well as file formats seems to be integrated. Such coupling is very tight and limits its applicability. For SQuADT we set the target higher. We have chosen to avoid building in knowledge about particular tools or even file formats.

Other approaches we know of are the electronic tool integration platform (ETI) [TRB05] and repository ([IMA$^+$07]). Both are built around web-services technology (using SOAP [ea03] and WSDL [CCMW01]). The tools either are a web service or wrapped inside a web service which allows loosely connect tools in a way very similar to ours. In ETI a tool can be connected by means of filling in a web-form that generates an XML file that represents the tools' interface. This is very similar to the XML formatted message on tools capabilities. Connected tools are aware of the integration context and ETI offers facilities that are usable to tools via Java-specific remote procedure calls.

Repository also uses web-services but in contrast to ETI, the connection between tool and framework is through specialised scripts. This puts it somewhere in between Eucalyptus and ETI. The tools are not aware of the integration context so a script is needed to make a tool behave properly in the integration context.

For all of the above approaches it seems that tools communicate through files. A serious consideration on our side was that files could grow very big and that you do not want to copy those files unnecessarily across a network. This does not necessarily preclude the use of web-services as interface between tools and an integration framework; but it does not make it the most logical candidate either. Truthfully the use of web-services was not considered until a protocol implementation was already available.

References

[ACDP06] A. Malhotra, C. M. Sperberg-McQueen, D. Peterson, and P.V. Biron. XML schema 1.1 part 2: Datatypes. Technical report, W3C, February 2006. http://www.w3.org/TR/2006/WD-xmlschema11-2-20060217/.

[CCMW01] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web services description language (WSDL) 1.1. W3C Note, World Wide Web Consortium, March 2001.

[CFT+06] C.M. Sperberg-McQueen, F. Yergeau, T. Bray, J. Paoli, and E. Maler. Extensible markup language (XML) 1.0 (fourth edition). W3C recommendation, W3C, August 2006. http://www.w3.org/TR/2006/REC-xml-20060816.

[DZ83] J.D. Day and H. Zimmermann. The OSI reference model. *Proceedings of the IEEE*, 71(12):1334–1340, 1983.

[ea03] M. Gudgin et al. SOAP version 1.2, part 1: Messaging framework, W3C recommendation. www.w3.org/TR/2003REC-soap12-part1-20030624/, June 2003.

[Ecl] Eclipse foundation. http://www.eclipse.org.

[FB96] N. Freed and N. Borenstein. Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies. RFC 2045 (Draft Standard), November 1996. Updated by RFCs 2184, 2231.

[GMR+07] J.F. Groote, A. Mathijssen, M. Reniers, Y. Usenko, and M. van Weerdenburg. The formal specification language mCRL2. In Ed Brinksma, David Harel, Angelika Mader, Perdita Stevens, and Roel Wieringa, editors, *Methods for Modelling Software Systems (MMOSS)*, number 06351 in Dagstuhl Seminar Proceedings. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany, 2007.

[IMA+07] I.G.J. Raedts, M. Petkovic - Ilic, A. Serebrenik, J.M.E.M. van der Werf, L. Somers, and M. Boote. A software framework for automated verification. In Y. Cho, R.L. Wainwright, H. Haddad, S.Y. Shin, and Y.W. Koo, editors, *Proceedings 22nd Annual ACM Symposium on Applied Computing (SAC 2007, Seoul, Korea, March 11-15, 2007)*, pages 1031–1032, New York NY, 2007. ACM Press.

[JHA+96] J.C. Fernandez, H. Garavel, A. Kerbrat, L. Mounier, R. Mateescu, and M. Sighireanu. CADP: a protocol validation and verification toolbox. In Rajeev Alur and Thomas A. Henzinger, editors, *Proceedings of the Eighth International Conference on Computer Aided Verification CAV*, volume 1102, pages 437–440, New Brunswick, NJ, USA, 1996. Springer Verlag.

[MD02] M. Mealling and R. Denenberg. Report from the Joint W3C/IETF URI Planning Interest Group: Uniform Resource Identifiers (URIs), URLs, and Uniform Resource Names (URNs): Clarifications and Recommendations. RFC 3305 (Proposed Standard), August 2002.

[Net] Netbeans community. http://www.netbeans.org.

[Pos81a] J. Postel. Internet Protocol. RFC 791 (Standard), September 1981. Updated by RFC 1349.

[Pos81b] J. Postel. Transmission Control Protocol. RFC 793 (Standard), September 1981. Updated by RFC 3168.

[Res01] P. Resnick. Internet Message Format. RFC 2822 (Proposed Standard), April 2001.

[SA04] P. Saint-Andre. Extensible Messaging and Presence Protocol (XMPP): Core. RFC 3920 (Proposed Standard), October 2004.

[TRB05] T. Margaria, R. Nagel, and B. Steffen. Remote integration and coordination of verification tools in jETI. In *Proceedings of 12th IEEE Int. Conf. on the Engineering of Computer Based Systems*, 2005.

[YPB+06] F. Yergeau, J. Paoli, T. Bray, C. M. Sperberg-McQueen, and E. Maler. Extensible markup language (XML) 1.0 (fourth edition). W3C recommendation, W3C, August 2006. http://www.w3.org/TR/2006/REC-xml-20060816.

A Appendix

A.1 Behavioural model

The following listing shows an mCRL2 model of the combined communication behaviour of the protocol. Besides communication tool start and termination only communications actions are visible. The model features a single controller and a single tool.

```
% Type that represents a configuration
sort Configuration = struct configuration(valid : Bool);

% Type used for identifying messages sent by the controller
sort MessageType = struct
   identification |    % identification response
   capabilities |      % request/respond for capabilities
   configuration |     % tool configuration specification
   task_start |        % signal that a tool may start task execution
   task_stop |         % signal that a tool has stopped task execution
   display_layout |    % communicates a display layout
   display_data |      % communicates state changes of the display
   termination |       % request/response for tool the termination
   report;             % report

sort MessageData = struct empty | result(Bool) | other;

sort Address = Nat;

% Messages travel between a controller (address 0) and a tool instance (address 0 <)
sort Message = struct message(type : MessageType, data : MessageData);

act

   send, receive, communicate: Address#Message; % send or receive a message
   start_tool, tool_start: Nat;                 % create a new tool instance
   terminate_tool,tool_terminate: Nat;          % terminate a tool
   execute_task: Configuration;                 % perform the configured operation
   error;                                       % error action
   update_display, waiting;                     % other non-communication actions

% a: the tool id
```

```
proc unidentified_tool(a : Nat) =
      send(a, message(identification, other)).
        tool(a, configuration(false), false, false);

% a: the tool id
% c: configuration object
proc tool(a : Nat, c : Configuration, in_configuration : Bool, in_task : Bool) =
   terminate_tool(a).delta +
   receive(a, message(termination, empty)).          % received termination request
     send(a, message(termination, other)) +           % sends termination signal
   (receive(a, message(capabilities, empty)).         % received request for capabilities
      send(a, message(capabilities, other)) +         % response capabilities (tool)
    send(a, message(capabilities, empty)) +           % requested controller capabilities
    receive(a, message(capabilities, other)) +        % response capabilities (controller)
    send(a, message(display_layout, other)) +         % sent display layout
    send(a, message(display_data, other)) +           % sent display data
    receive(a, message(display_data, other)) +        % sent display data
    send(a, message(report, other))).
   tool(a, c, in_configuration, in_task) +
   (!in_configuration) ->
     ((!in_task) ->
       receive(a, message(configuration, empty)).
         tool(a, c, true, false)) <>                  % received a task specification
     (send(a, message(configuration, other)) +
       send(a, message(configuration, empty))).
         tool(a, configuration(true), false, false) +   % accepted task specification
   (!in_task) ->
     receive(a, message(task_start, empty)).          % task start signal
     tool(a, c, in_configuration,
             (valid(c) && !in_configuration)) <>
     (sum b : Bool.
       send(a, message(task_stop, result(b))).        % task finished or aborted
       tool(a, c, false, false));

% Workaround until proper support for finite sets is implemented
map set_add    : Nat # List(Nat) -> List(Nat);
    set_erase : Nat # List(Nat) -> List(Nat);

var x,xx : Nat;
    xs   : List(Nat);

eqn set_add(x,[]) = [x];
    set_add(xx,x|>xs) = if(xx in xs,xs,if(xx < x,xx|>x|>xs,x|>set_add(xx,xs)));
    set_erase(x,[]) = [];
    set_erase(xx,x|>xs) = if(xx == x,xs,x|>if(xx in xs,set_erase(xx,xs),xs));

% m: the maximum address of a running tool
% r: the set of active tools
% c: the set of active tools that are configured
% s: the set of active tools that are executing a task
proc controller(m : Nat, r : List(Nat), c : List(Nat), s : List(Nat)) =
 (sum a : Nat. (a <= m) -> (
   start_tool(a).controller(m,set_add(a,r),c,s) +   % starts a new tool instance
   (a in r) -> (                                    % for tool a:
     receive(a, message(identification, other)).    % received instance identifier
       controller(max(m,a),r,c,s) +
```

```
    (receive(a, message(capabilities, empty)).     % received request for capabilities
       send(a, message(capabilities, other)) +
     send(a, message(capabilities, empty)).        % sent request for capabilities of
       receive(a, message(capabilities, other)) +
     receive(a, message(report, other)) +          % report delivered
     receive(a, message(display_data, other)).     % update for display data
       update_display +
     send(a, message(display_data, other))).       % data from user interaction
    controller(m,r,c,s) +
    send(a, message(configuration, empty)).        % configuration request
      controller(m,r,set_erase(a,c),set_erase(a,s)) +
    send(a, message(termination, empty)).          % sent termination command
    receive(a, message(termination, other)).       % termination signal
     terminate_tool(a).
     controller(m,set_erase(a,r), set_erase(a,c),
                  set_erase(a,s)) +
    receive(a, message(display_layout, other)).    % display layout delivered
      controller(m,r,c,s) +
    receive(a, message(configuration, other)).     % configuration was accepted
      controller(m,r,set_add(a,c),s) +
    receive(a, message(configuration, empty)).     % configuration was rejected
      controller(m,r,c,s) +
   (sum b : Bool.
     receive(a, message(task_stop, result(b)))).
     ((a in s && a in c) ->
        controller(m,r,c,set_erase(a,s)) <>
        error.controller(m,r,c,s)) +               % task completed
   ((a in c) ->                                    % for tool a:
     send(a, message(task_start, empty)).
       controller(m,r,c,set_add(a,s))))));         % task start command

% Execution component (starts tools)
proc E(current : Nat, maximum : Nat) =
  (current < maximum) ->
        start_tool(current).E(current + 1, maximum);

init

  hide({error, waiting, execute_task, update_display,tool_terminate},
    allow({communicate, tool_start, tool_terminate,
                        error, waiting, execute_task, update_display},
      comm({send|receive -> communicate,start_tool|start_tool -> tool_start,
            terminate_tool|terminate_tool -> tool_terminate},
        controller(0,[],[],[]) || unidentified_tool(0) || E(0, 1))));
```

Message content is abstracted to that portion that is relevant for communication. In this way we distinguished three classes of messages, those with empty contents, those with a true/false result and all of the others. For example, requests characterised by the message type and empty contents.

A.2 Graphical Representation of Communication Behaviour

The following figure shows a graphical representation of the communication behaviour using the mCRL2 model presented previously. For presentation purposes the communication actions have been replaced by the names of the message they communicate. The picture was generated with the ltsgraph tool after instantiating the state-space and and minimising modulo branching-bisimulation. The final result took some manual polishing.

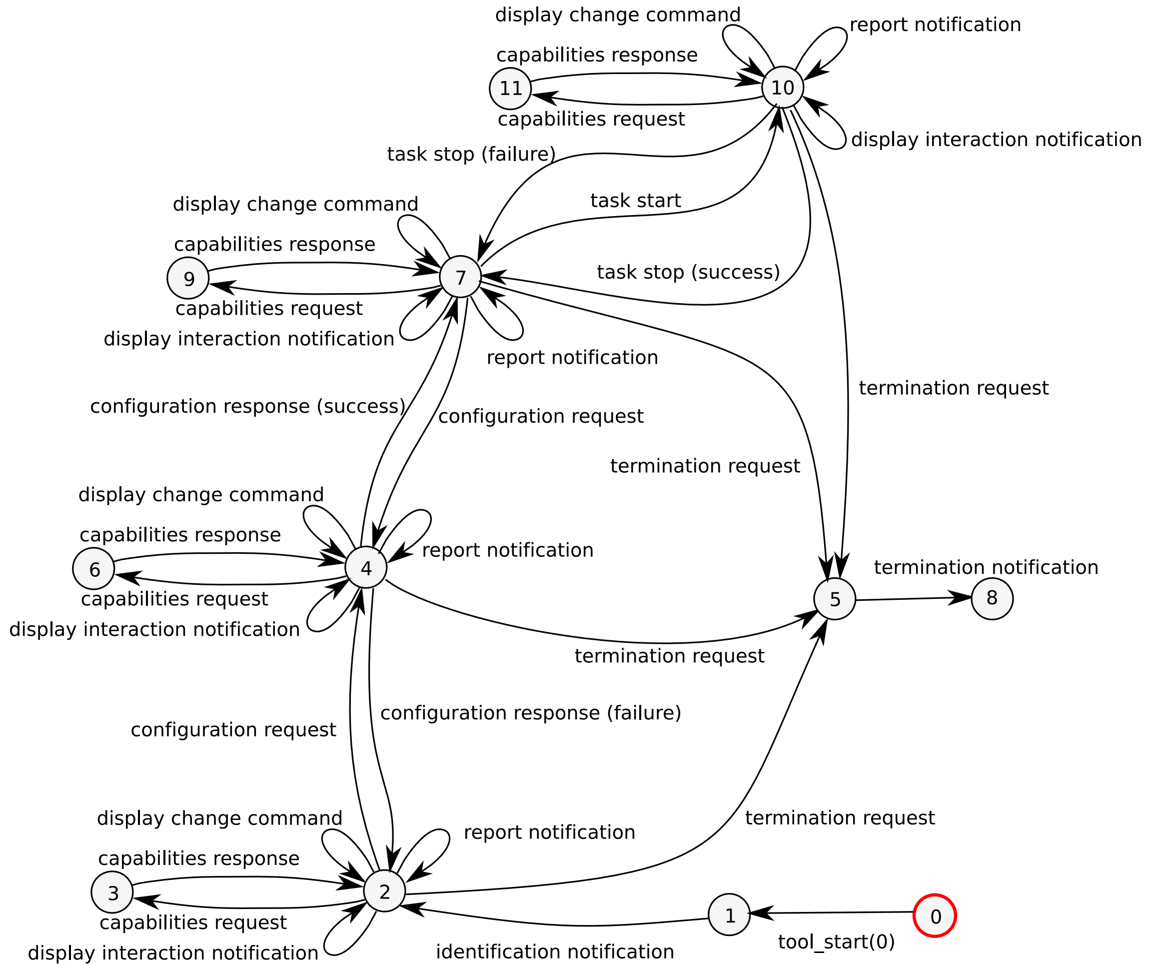


Figure 9: Graphical representation of the state-space

The constraints as they are found in section 3 together make up the behaviour depicted above. What is visible is the intended pattern of communication. A protocol implementation must abort with an error condition for message sequences outside those allowed by the model.

A.3 Example Layout

Below is a listing for the layout graphically depicted in figure 8.

```
<display-layout>
 <layout-manager>
  <box-layout-manager variant="horizontal" id="top">
   <box-layout-manager variant="vertical" id="top_top">
    <box-layout-manager variant="vertical" id="top_top_left">
     <properties margin-left="5" margin-top="10">
     <label id="alabel"><![CDATA[This is a label]]></label>
    </box-layout-manager>
    <box-layout-manager variant="vertical" id="top_top_right">
     <properties margin-left="0" margin-top="0" horizontal-alignment="center">
     <box-layout-manager variant="vertical" id="top_top_right_top">
      <properties horizontal-alignment="center">
      <button id="abutton"><![CDATA[B Button]]></button>
     </box-layout-manager>
     <box-layout-manager variant="vertical" id="top_top_right_bottom">
      <properties horizontal-alignment="left">
      <button id="abutton"><![CDATA[A Button]]></button>
     </box-layout-manager>
    </box-layout-manager>
   </box-layout-manager>
   <box-layout-manager variant="vertical" id="top_bottom">
    <properties horizontal-alignment="center">
    <progress-bar id="progress" minimum="0" maximum="1000" current="350" />
   </box-layout-manager>
  </box-layout-manager>
 </layout-manager>
</display-layout>
```