Development of a Visual Refinement- and Exploration-Tool for SpecC

David Berner
Prof. Dirk Jansen
Prof. Daniel D. Gajski

Diplomarbeit
February 22, 2001

Center for Embedded Computer Systems
University of California, Irvine
Irvine, CA 92697-3425, USA

mail@davidberner.de
http://www.cecs.uci.edu/~berner

Development of a Visual Refinement- and Exploration-Tool for SpecC

David Berner
Prof. Dirk Jansen
Prof. Daniel D. Gajski

Diplomarbeit
February 22, 2001

Center for Embedded Computer Systems
University of California, Irvine
Irvine, CA 92697-3425, USA

mail@davidberner.de
http://www.cecs.uci.edu/~berner

Abstract

This document describes the development of RESpecCT, a refinement and exploration-tool for the SpecC technology. RESpecCT is a graphical tool which assists the designer starting from the functional or specification model of the design in refining it using the SpecC methodology through different levels to the implementation- or register transfer-level. It visualizes information in a way to simplify the process of taking decisions about details of the design, gives these decisions to different refinement tools and visualizes their results.

Erklärung

Ich versichere, dass ich diese Diplomarbeit selbständig verfasst und keine anderen als die angegebenen Hilfsmittel verwendet habe.
Irvine (USA), den 22. Februar 2001

David Berner

Acknowledgment

I would like to thank Prof. Dr. D. Jansen from the Fachhochschule Offenburg and Prof. Dr. D. Gajski from the Center of Embedded Computer Systems, University of California Irvine for acting as advisors for this thesis. They gave this great opportunity to do it abroad in this marvelous environment.

Many thanks to Andreas Gerstlauer who was always there for questions and discussions (in English as well as in German).

Thanks also to all the members of the SpecC group with whom i tried hard to do good work in order to establish SpecC as a world standard.

Thanks to Judith who waited all this time so patiently for me.

Contents

List of Figures

1 Introduction

In 1965 Gordon Moore (co-founder of Intel) predicted that the transistor density of semiconductor chips would double roughly every 18 months. In February 2001, Intel's Chief Technology Officer Pat Gelsinger pointed out in the opening speech of the International Solid State Circuit Conference (ISSCC) in San Francisco that by the end of this decade processors will reach 1 TIPS (Tera Instructions Per Second) at 30 GHz. These processors will consist of about 10 billion transistors.

A current Intel Pentium III 1 GHz processor (codename coppermine) has about 28 million transistors, two third of which actually represents the 256 kB on-die level 2 cache. The increase of designer productivity measured in the number of processors designed has been only about 20% per year in the recent past. If we project the same increase over the following nine years, in 2010 a processor would have less than 0.2 billion transistors.

So how are we going to design the missing 9.8 billion transistors? Are we going to employ 50 times the number of designers? Not likely. Certainly it is more desirable to increase the productivity of the designers drastically. But how?

One possible answer is to use designs, that are highly modular and easily reusable in later generations of products - IP-reuse. Another answer is to work at a higher level of abstraction.

1.1 System-level design

The highest level of abstraction is to perform synthesis directly from the system specifications. This approach is called System Level Design (Figure 1). It means to write a system specification in a language as intuitive as possible and to refine this specification down to the register transfer-level (RTL).

SpecC is a language (Section 2) and a methodology (Section 3) especially designed for system level design. It's basic concept is not the one-button-push solution, but an assisted refinement process. It defines several level of refinement at each of which the designer is provided with information in order to help him taking decisions for the next refinement step. After the last refinement step the design is at RTL and synthesizable.

1.2 Goal

In the SpecC refinement-process there are several tools involved (Profiler, Architecture Refinement Tool, Communication Refinement tool, SpecC compiler, etc.), but the user only wants to have to deal with one. So we wanted to create a graphical user-interface which integrates all tools and assists the designer during the whole refinement-process - from specification to transistor-level.

It was clear that this is a big task and that it would take several man-years to accomplish it, so given the actual time-constraints, we focused on the specification of the project and the choice of the tools and environment. Then it was important to create a stable and open basic framework which demonstrates the basic concepts and would be easily understandable and expandable.

1.3 Related Work

SpecC is quite unique and there is still done research, so there is nothing which closely relates to it. Sure there are several other approaches to System Level Design such as System C or VCC, but their concepts are

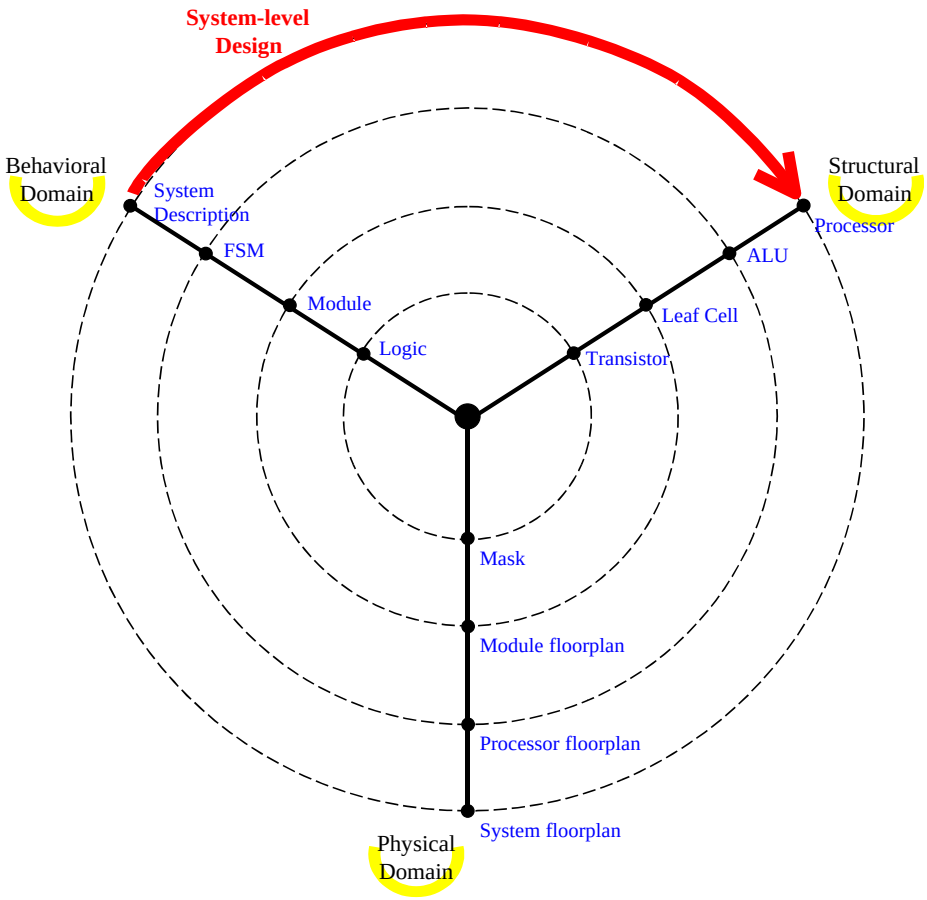


Figure 1: System-level Design in the Y-Chart.

totally different.

The Toshiba Corporation has made a tool called VisualSpec which is a GUI for SpecC. VisualSpec incorporates some of the functionality mentioned above, but it is not at all what we were anticipating. This is why we started with this project from the beginning and called it *RESpecCT* (Refinement and Exploration-tool for the SpecC Technology).

2 The SpecC Language

SpecC is a language.
It is not a language grown over centuries in order to adapt to constructs people like to express - like most of the human languages, nor has it been developed and then expanded and adapted to different applications people may have - like most of the programming languages. SpecC is a language developed for one special purpose: system level design.

The main properties of SpecC are that it has an easily understandable syntax for machines as well as for humans, it includes all constructs needed to describe a design as inherent parts of the language and it can be used to describe a design at all levels of development - from specification to register transfer level.

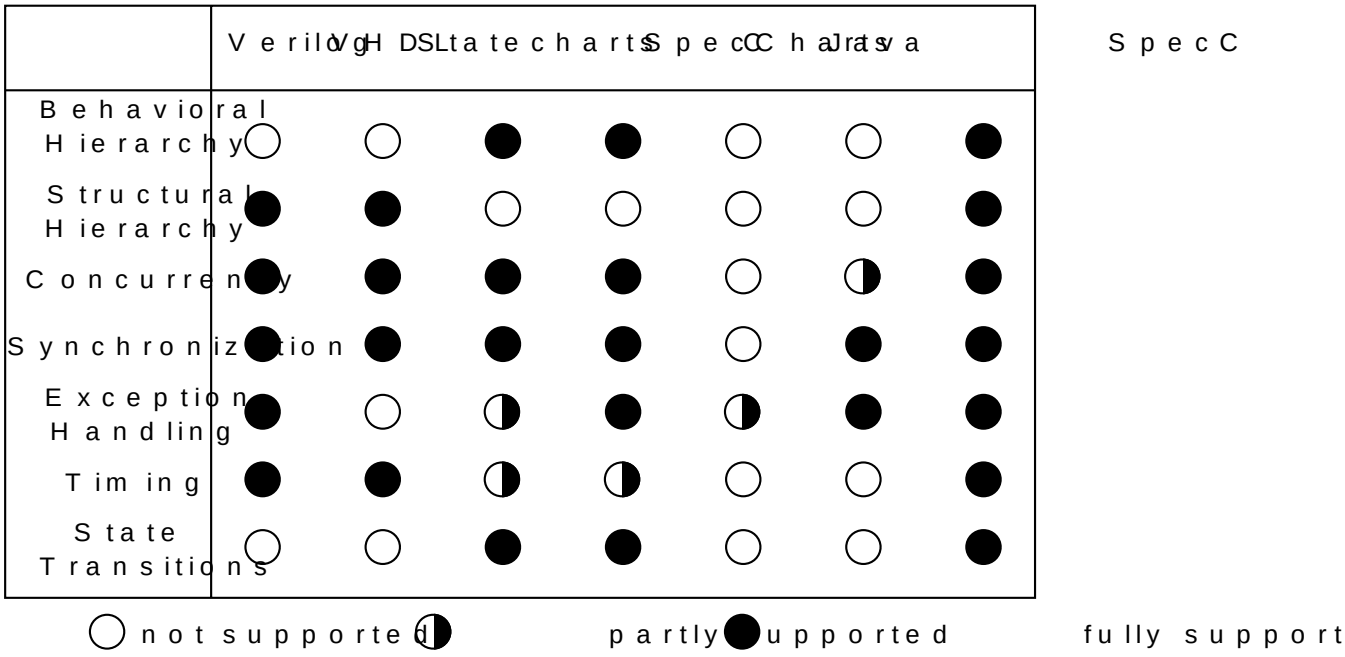


Figure 2: Language Comparison [1].

2.1 C+Spec = SpecC

The reason for creating SpecC as a new language was because there was no such thing. SpecC was started an a methodology (Section 3) and then Gajski and his group were looking for a language which could be used to implement this design-flow best. Languages like C and Java are very well suited to write a functional model. They have also the advantage, that many people know them so there is not much training necessary to get people productive. On the other hand, these general purpose programming languages lack functionality when it comes to the refinement. How am i going to describe e.g. parallelism in C? So if we want this to work, we have to write additional libraries that add certain functionality to the language. This is possible, but it

gets somewhat inconsistent. System C e.g. takes this library-based approach. An advantage is that tools like compiler, debugger and development environment already exist. On the other hand describing mechanisms like timing and concurrency is not very intuitive by using a library. Also, a library based approach is easy to expand. This can be seen as an advantage e.g if a company wants to adapt it closer to their application, but since everyone keeps changing it that much, it is impossible to have a set of tools handling all the constructs of the language.

An other approach is to take existent design languages like VHDL and Verilog and try to fit them into the whole design process. This approach turns out to be difficult, too since these languages are just not made for high level descriptions. A functional description of a system in VHDL would either be already mapped to certain architectures or would require to expand VHDL which would result in a new language as well.

SpecC tries to incorporate the advantages of both worlds while excluding the drawbacks. It uses ANSI-C as a basis which is well known, easily understandable and well suited for functional descriptions. It adds few constructs for handling missing functionality for the design flow like concurrency and hierarchy (Section 2.2). The choice of the added statements once done properly, it will be very hard to change, since the compiler and all tools depend on it. This makes SpecC easily to standardize.

2.2 Special Features

SpecC adds only seven major mechanisms to C. These are :

- Behavioral hierarchy
- Structural hierarchy
- Concurrency
- Synchronization
- Exception handling
- Timing
- State Transitions

With the help of these mechanisms we are able to write a functional description of the design which can then be successively refined to a cycle accurate description of the system. Details about particular constructs can be found in the SpecC book [**?**]. There are also some small additions not mentioned here e.g. a true boolean type. Details about those can be found in the SpecC Language Reference Manual [**?**].

2.3 Summary

The SpecC language was designed because there is no other language which fits - or can be fitted with reasonable effort - into the SpecC methodology. It is executable on every stage of refinement, it is highly modular because of the behavioral concept, supports design reuse and is complete in terms of supporting all concepts currently used in embedded systems. All it’s concepts are organized orthogonally, which results in a more or less minimal solution and makes the use of the concepts consistent. As ANSI-C is used as a basis, is is easy to understand and to learn.

3 The SpecC Methodology

3.1 Overview

The SpecC methodology defines the process to get from a functional system specification to an implementation on register transfer level (RTL) which can be given to a fab in order to actually produce the chip. The methodology comprises four system-description models and transformations between these models (Figure 3). Following the transformations, the systems move from the specification model over the architecture model and the communication model to the implementation model. Each model represents a new level of refinement, introduces new concepts and properties into the design.

The main idea is to have a design with is simulateable at every stage of development and which - once specified - does never have to be rewritten during the whole process, resulting in a failsafe and consistent design.

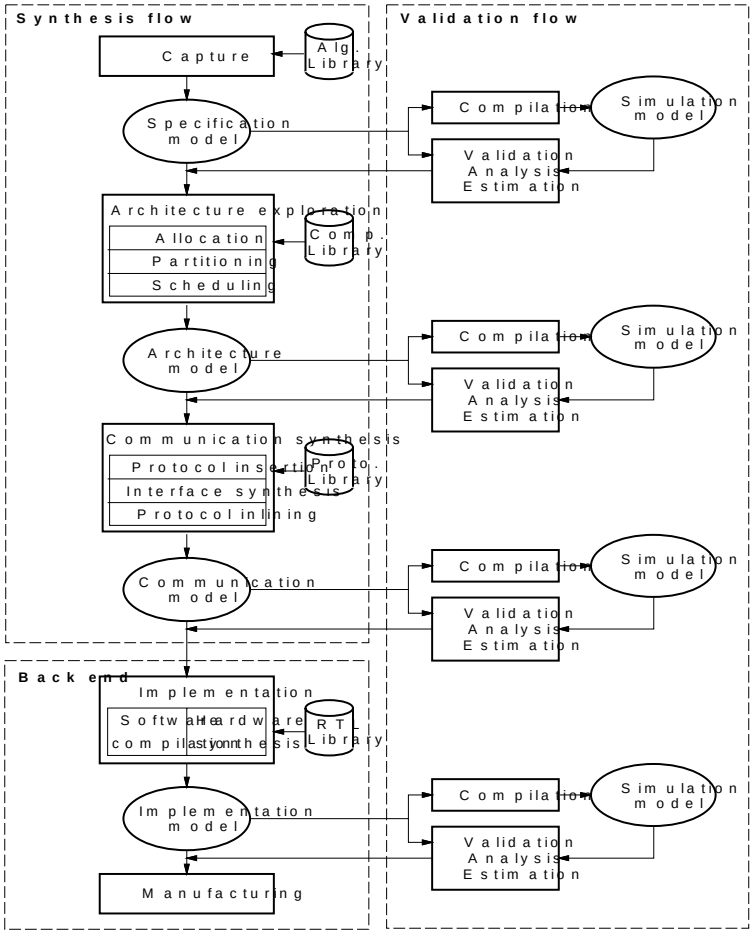


Figure 3: The SpecC Methodology [1]

3.2 Specification Model

The highest level of abstraction in system design is the functional description of the system. It describes the whole system in detail but does not include implementation-details like timing or partitioning. While in theory every functional description results in a valid design, it does not forcedly result in a ”good” implementation. In order to get efficient results, one should stick to some simple rules in describing the functional model:

1. Separate communication and computation A basic idea in SpecC is to separate computation from communication. This makes it easier afterwards to try different protocols without touching the computation part. In SpecC the computation is specified in behaviors, whereas the communication is contained in channels. Input and output of behaviors have to be specified explicitly in order to be able to evaluate data dependencies.

2. Expose parallelism
   Without knowing how many processors you will use in the design and of which kind they are, the specification model should expose as much natural parallelism as possible. Modeling behaviors in parallel - if they are not dependent of each other - will give more space to optimizations during refinement. To serialize behaviors which are modeled in parallel is not a complex task whereas to detect if a group of serial modeled behaviors can be run in parallel is much more extensive.

3. Use hierarchy to reflect functionality
   For every design of considerable size it is most important to structure it reasonably so the designer can keep an overview over the whole project while being able to locate details in the design. A very common and convenient way to do this is to use hierarchy. Now if the concept of hierarchies is used wisely, the design stays both, easily understandable and easy to optimize and refine.

4. Choose proper granularity
   Since the whole design is divided into behaviors, one question is then what to put in one behavior, and where divide it into several. If the size of the behaviors is chosen too big, the refinement will not be optimal. Since most optimizations are done on the behavior-level, the behaviors themselves remain more or less untouched. If the size of the behaviors is chosen too small, the design complexity will be high. As a hint, behaviors should always represent basic algorithmic blocks. This will leave enough room for optimization while leaving the design complexity on a tolerable level.

5. Identify system states

Following these rules, the design will be suitable for refinement and will profit as much as possible from the advantages of the SpecC methodology.

3.3 Architecture Exploration

Once the system is defined, the design gets into the refinement-phase. The first transformation is called ”architecture exploration”. It transfers the design from the specification level to the architecture level.

In the Architecture level, the architecture of the design is known. This means that the number and the type of the processors is fixed. In addition one has already fixed what part of the design will be run on what processor and in what order. To get to the architecture model there are basically three steps to be performed:

1. Allocation
   During allocation the designer selects the number and type of processors to be used. To take these decisions he has to evaluate the design in respect of size, speed, performance constraints, cost and more. This data he can use in order to select processors from an IP-database where some additional data is available.

2. Partitioning
   Having decided which processors to use, one has to decide which part of the design will run on which processor. This process is called partitioning or behavior-mapping. Again, this is no easy step. To perform it, the designer needs information about what loads different parts of the design cause. Also he wants to know how much memory the behaviors use. Very important is the communication of behaviors with other parts of the design. It is not advisable e.g. to put parts of the design on different processors which communicate a lot between each other. This would cause much traffic on the bus which connects them and is usually unwanted. Only in very special cases one could want to do this in order to avoid other problems.

3. Scheduling
   If all behaviors have been mapped to processors, the order they execute is still to be fixed. This is easy for behaviors which have already a serial decomposition, but can be hard for parallel and especially for FSM-behaviors. With the scheduling there will be synchronization behaviors introduced. They manage synchronization e.g. for originally parallel behaviors.

After architecture exploration we can make more accurate predictions about performance and cost. We will know quite accurate execution-times since we know exactly the processor on which the code will run and can make a very accurate simulation. On the Architecture Level there will be an additional level of hierarchy which represents the components. Also there will be additional behaviors on this level synchronizing the communication between these processors.

3.4 Communication Synthesis

Communication synthesis is the transformation from the architecture level to the communication level. While on architecture level the communication between the components is still done in channels - which means that it happens in no time (or predefined intervals) - on the communication level the processors communicate via wires on a defined bus with a defined protocol. In order to perform the communication synthesis two steps have to be accomplished:

1. Bus allocation
   For the bus-allocation one has to select one ore more busses out of an IP-database. They have to be selected considering parameters like bandwidth, bit-width, cost, overhead and the protocol.

2. Channel mapping
   After selecting the busses, the top-level channels have to be assigned to the busses. If the communicating processors do not support the bus-protocol, there have to be inserted additional components which take care of the conversion of the different protocols. We call them transducers.

In the communication model we have not only correct execution times of behaviors, but we have a cycle accurate model of the communication part of the design. Simulation will now reveal all timing problems that were still undetected in the design.

3.5 Summary

The SpecC Methodology is a design methodology specifically tailored for system level synthesis. It starts with a functional system description of the design and will be transformed over several steps to a cycle accurate design description ready for implementation. The critical decisions in the refinement process have to be taken by a designer while straightforward - but still complex - refinement tasks are automated by tools.

This has the advantage that the design process is highly automated, but the designer still has the control over the refinement process so the results are very efficient.

4 Specification

Before actually start working on a project, it is essential to specify as exactly as possible what one really wants to have. At the very beginning it is not easy to specify the needs, also the specification has to be reasonable in terms of resources like time and money.

In a first task-analysis we investigated what actually had to be done, then we set priorities to get an order what had to be done first and what was less important. Then we refined the results of the task-analysis in order to obtain a quite precise specification.

4.1 Edit

An evident property of RESpecCT should be to read designs of different formats, convert them to others and write them out again. The user wants to be able to browse the hierarchy of the design, to get provided with all kinds of information about the design as well as about parts of the design or about single behaviors. We want the user to be able to add change and delete behaviors or properties of behaviors like variables, channels and ports. Some of these actions may be quite complex, there may be numerous dependencies and prerequisites. In these cases the user should be assisted through the whole process with wizards.

In addition to a graphical display there should also be support to directly edit the code for people who want to do quick changes in the code without having to deal with the GUI. However, these changes should be immediately reflected in the graphical representation in order to keep consistency between these two.

Apart of the hierarchical view, there should be a display of the connectivity inside behaviors. It could look like a matrix with ports of the sub-behaviors over the connected variables, channels and ports. It will be a sparse matrix with one connection per column. The user should be able to make or release connections between items.

4.2 Project Management

A SpecC-design typically consists of several files. In order to keep the work organized, it is essential to be able to define a projects which includes all files and keeps track of their location. Such a design project could store additional information like descriptions and comments for the whole projects as well as for every file. Since SIR-files are binary and thus not portable over platforms, it is important to always keep recent versions of SpecC-files.

As soon as we start with the refinement, the projects get an additional meaning. It keeps files of all four design-levels (Specification level, Architecture level, Communication level and Register Transfer Level) in order to be able to go back in the design-process, change decisions and repeat already performed steps. This is very important, since during refinement behaviors are added to the design (e.g. for synchronization) and changes are made to the hierarchy without being able to undo this. The reason for this is, that different specifications can theoretically end up in the same architecture-level design.

The file-management also should support the user to keep consistency throughout all design-levels. RESpecCT throws a warning if the user tries to modify the design on a level other than the specification level. If he does anyway, consistency between the different levels is no longer given. A correct simulation of a higher level does then no longer mean that a lower level simulation of the design has to be correct. A big advantage of the SpecC-methodology would be lost. The correct way to do changes in the design is to go

back to the specification level - without losing the information of previously takes decisions - perform the changes and then redo the refinement.

Another job of the project-management is to keep options like selected weight-tables, profiling options and the name of the top-level behavior.

4.3 Build

At each time of the design-process, SpecC makes sure, that the design is simulateable. In order to perform simulation three steps are to be done: Compilation, Debugging and the actual Simulation.

4.3.1 Compiler

The compilation of the design converts the SpecC-code into an SIR data-structure, then converts it into C++ code which is then compiled into an executable. The SpecC-compiler should take care of all this.

4.3.2 Debugger

During the process of writing the specification of the design, it is very likely that the design is not correct from the beginning. Therefore we have to provide a possibility to debug the design. Since it is actually the C++-code not the SpecC-code which is complied, the actual line numbers reported by the debugger have to be translated to the SpecC-line numbers. This should happen transparently so the user is not aware of that.

For more convenience it should be possible to set breakpoints in the code and to trace over it or step into it.

4.3.3 Simulation

Running the actual simulation is pretty simple. The executable created by the compiler has to be executed. However, it has to be made sure that there is a correct testbench around the design. Also the testbench could require parameters which have to be provided and must be adjustable by the user. Last but not least, the user wants to get feedback about the simulation in the interface. If an error occurs it could indicate a wrong parameter or a bug in the testbench. Also the regular output of the simulation alone could be instructive.

4.4 Profiling, Estimation

The main purpose of profiling and estimation is to get information out of the design which helps in taking further design-decisions e.g. about what processors to use, what memory-size will be needed, what busses are suitable and to find out at a very early stage if the design fulfills certain constraints like price, speed or size.

Although these two mechanisms seem very similar in their goal, they do totally different things. This is also due to the fact that they are performed at different stages of the design.

Profiling can be performed at the very beginning. As soon as there is a specification which is semantically and syntactically correct and an executable testbench, one can run the profiler. Like this we are able to examine certain properties of the design while it is still growing. Fundamental errors can be detected early and taken care of, without wasting too much time and money.

The profiler inserts statements into the design which produce information about operations performed in the design at simulation-time. After simulation it evaluates this information and makes it available. Profiling information comprises operations, memory-usage and communication-intensity on a per-behavior basis. For every behavior we got then a lot of information. There are 65 different operations two different categories for memory and for communication (in and out). Every category is divided into the 29 different data types. This makes 2001 pieces of information per behavior. In addition the profiler extracts some statistical metrics out of it. All this information has to be available to the designer in the GUI in order to evaluate it. RESpecCT should offer some general high-level metrics by default, but is able to deliver more specific information on request.

The profiler also should support the mechanism of weight-tables. Weight tables are used to reflect properties of the the actual architecture better in the results of the profiling. For example if the processor we anticipate to use has a very slow multiplication-unit, multiplication-operations will be counted e.g. three times, whereas additions are counted only once. Then, if the profiling counts 3 multiplications and 3 additions for a certain behavior, the number of total operations would be 12. This kind of information is stored in a weight table. It makes the profiling produce more accurate results if we want to add up different kinds of operations.

Estimation is a mechanism which can be used only after architecture exploration. While the profiler does a dynamic analysis of the code, the estimator does only a static analysis. Together with the data from the profiler and the results of the architecture exploration, it can provide more accurate values than the profiler. Also you get absolute information about time of execution, while profiling only provides relative information about speed and performance.

Profiler as well as Estimator are separate tools written in C++. They have to be integrated into RESpecCT as a shared library. The data-exchange should be done entirely within the SIR-datastructure.

The sensitive point about profiling and estimation is to extract the right judgments out of the whole bunch of information they provide. In order to optimize that, we have to develop clear and intuitive displays. Displays which can display a lot of information at a time while emphasizing on the critical points. Since there is never one display that can show all aspects, there have to be several which complement each other.

These displays could be:

- Columns in the behavior tree
- Pie chart
- Bar chart
- Table

4.5 Architecture exploration

As we already know, architecture exploration is the process of mapping the specification model to a certain architecture. This includes insertion of components, and the synchronization between these.

4.5.1 Allocation

The first step in the process of architecture exploration - consists of selecting one or several processors from a database. In order to provide the user a basis of comparison, we have to list the processors with some short description as well as some details like performance, clock, area and cost.

Also we should be able to add and remove processors at any time. One processor can be selected several times for same design, so the user has to assign them unique names.

4.5.2 Partitioning

During this phase the behaviors get mapped to the processors allocated. It is a quite critical step of the refinement-process and has to be done carefully. There are some simple rules to respect. For example one will always try to separate two blocks which are as independent as possible. One should always try to leave behaviors together which communicate heavily with each other.

Also it can be advisable to separate behaviors which mainly perform the same kind of operations. If there is a group of behaviors which seem to use mainly 32 bit multiplications, we could try to map them together to a processor which is optimized for that.

After partitioning is completed - which means all behaviors except the ones belonging to the testbench are mapped to a processor - one can run the architecture refinement tool. It will reorganize the hierarchy, introduce a new level of hierarchy representing the processors and add behaviors which manage the synchronization between the processors.

4.5.3 Scheduling

Only after partitioning is completed and the refinement-tool has processed the data-structure, we can start to do the scheduling. Scheduling means to assign an order to the behaviors on each processor. This is easy for a serial behavior, but can be critical for parallel or other behaviors e.g. FSM. There the user has to be aware of the dependencies between the behaviors and be able to interpret them correctly. The user can also ”flatten” parts of the hierarchy, that means do the scheduling not only of one level, but include behaviors from lower levels.

Though the GUI will display the scheduling decisions, the change in the data-structure will not be performed unless the user actually runs the refinement-tool again. After this step, there will be no more serial structures inside one processor. The design is now on the level of the Architecture model.

4.6 Communication synthesis

To get from the architecture model to the communication model, we have to do the communication synthesis. The procedure resembles the architecture refinement: First, there has to be done allocation, then mapping.

Allocation means basically to select the busses one wants to use in the design out of a database and give them a unique name. Like in the architecture refinement, there should be displayed some data with the busses like bits/sec, frequency or number of bits in order to let the user make a conscious choice, even if he is not familiar with all the busses.

For the mapping, all top-level channels have to be assigned to a bus. A top-level channel is a channel over which the processors communicate. There may be a lot of other channels in the design, but since usually

there are no busses inside of components, we do not have to worry about them. All the channels have to be mapped to a bus, and to every allocated bus has to be assigned at least one channel.

Once the mapping is completed, the communication refinement-tool can be called. It will again change the structure of the SIR. There will be suitable protocols inserted for the busses and the top-level channels will be converted into actual bus-wires. These will be represented as variables.

4.7 Summary

This chapter illustrates how much work is involved in the project and how a first version could look like. Even after all this has been implemented, one could think of a whole bunch of other functions, not mentioned so far.

To get the project started, we had to define a priority-list what the first implementation would look like and what will be targeted in the further development. Given the initial time-restriction of six months including the preparation-phase, we decided to concentrate on a basic framework. There we should integrate the SIR and prove it's usability. We should provide the basic widgets, show how to integrate an external module and how to run external tools. Also a reasonable documentation of both, the functionality and the code should be part of it.

Once this is done solidly, it should be relatively easy for someone else to continue the project by just using the concepts demonstrated in the first stage. Other modules can be integrated just in the same manner and if there is the need, widgets can be extended.

Stage 1 consists of the following tasks:

- Display the behavior-hierarchy and browse it
- Edit source code
- Show how to display and edit properties of behaviors (channels, variables, ports)
- Show how to run external tools (like compilation and simulation)
- Show how to integrate external modules like the profiler
- Create different widgets for displaying data (e.g. profiling results and make them easily usable)

Then in a second step, the following issues should be done:

- Allocation
- Map behaviors
- Run architecture refinement tool
- Schedule behaviors
- Allocate busses
- Map busses

- Run communication refinement tool
- Run estimator
- Display estimation results
- Add and remove behaviors
- Add and remove properties (channels, variables, ports)
- Make project-integration

It is obvious, that list is not complete. But it is enough to get an idea of what RESpecCT could look like in the future.

5 Choosing the Tools

Once the project is specified, we are to choose the tools which fit best for the anticipated task.
Since the SIR-library was written in C++ and C++ is an object-oriented language which is widely used, writing the GUI in C++ seemed to be the natural choice. Java would have been an alternative, but it would have added a lot of additional work and no real benefit except the native cross-platform-support. As there are a considerable number of usable graphical toolkits for C++, even for cross-platform development, we quit Java.

5.1 Different GUI Toolkits

The term toolkit in this document means a library for the development of graphical user interfaces (GUI). Since ANSI C does not include any windows-library by itself, it is essential to use a toolkit for the development of a windowed application with C or C++.

As mentioned there are quite a few graphical toolkits available for C++. We will give a quick overview of the most important ones. The Criteria under which we looked at the toolkits were (sorted by importance):

- Portability They must at least be available under Unix and Windows.
- Completeness
- Usability
- Documentation
- Number of users
- Look and feel
- Price

Toolkits which get clearly disqualified by one ore more of the criteria are not listed unless we consider them generally important.

- Microsoft Foundation Classes (MFC)
  MFC are the leader in the Windows-world. They are very complete, quite convenient to use, good documented and there are a lot's of people using it.

  MFC is commercial, and ships with the Microsoft Visual C++. Although this already costs money, it only provides Microsoft Windows support. A Unix-version of the library has to be purchased separately and is very expensive.
- Gnome
  Free, native support for X11
  No support for windows
- QT
  Very good and complete. Free (GPL) version for Unix, commercial version for MS Windows. Very good documentation, professional support. Graphical widget-designer available.

  The MS-windows version is quite expensive.

- wxWindows (gtk+)
  based on gtk+. Very nice and complete. Free (GPL). Good support for both, Unix and MS Windows. Graphical widget-designer available for $50.

  Documentation not complete. Widget-designer instable.

- VDK (gtk+)
  Good widget-set based on gtk+. Nice, not too stable widget-designer. Available only for Unix.

- tcl/tk
  A wide range of widgets available. Widely used.

  Not object-oriented, widgets look not very beautiful (old fashioned), no C++.

  - [incrTcl]
    Object-oriented Wrapper for Tcl.
  - tkinter
    Python -Wrapper for Tcl. Object-oriented, good documented. No C++

- gtk +
  Nice and complete widget-set. Only C, no C++.

- Visual Component Library (Borland VCL)
  No Unix-support

5.2 QT versus wxWindows

After examining a lot of toolkits, installing and trying some, surfing the web for information, reading related newsgroups and discussions, we came to the decision, that there are in fact only two toolkits to choose from: wxWindows and QT. Table 1 shows a close-up comparison between these two:

Criteria	QT	wxWindows
Portability	++(MS Windows, Unix, Linux)	++ (MS Windows, Unix, Linux, Macintosh)
Price	- ($2925)	++ (free, designer for $50)
Documentation	++ examples, tutorial	+
Completeness	++	+
Usability	+	+
Number of users	++	o
Look and feel	++	+

Table 1: Comparison of QT and wxWindows

Even though wxWindows and QT seem both to be a good choice, the table shows that QT leads the comparison in almost all categories. We came to the conclusion, that even though QT has a considerable price, it is probably cheaper to develop in QT than in wxWindows. Also the professional support and a large user-community, gave us confidence to choose QT.

5.3 PyQt

Having selecting QT as the toolkit of our choice, we realized that there is also Python bindings for QT. Having known Python already as a very powerful language for developing scientific applications, we decided this needed further investigation. PyQt revealed itself as a free python wrapper for the QT-library.

5.3.1 Advantages

- Price
  A C++ QT License for both Unix and MS Windows from Trolltech is today $2925. This includes one year support and free upgrades. Every additional year of support and upgrades is $910.
  PyQt is free for Unix as well as for MS Windows. Although it looks there may be charges for the Windows-version in the future, these will be insignificant in comparison to the C++ license.

- Speed of Development
  Anyone who has ever developed a program of considerable complexity in Python may have been amazed how fast the project was advancing. In comparison to C++ (or to any programming-languages we know) the syntax of Python is much clearer. Indentation is a syntax element, so the source code is a lot easier to read - this is especially helpful if one tries to understand what one has done a couple of weeks ago.
  This in conjunction with the fact that major changes in the program structure - which unfortunately occur especially in the beginning of application development - makes Python a very handy Rapid Application Development (RAD) -Tool. As the syntax - though a lot simpler - is very near to C++, one has always the option to rewrite the project in C++ once it reaches a more mature level.

- Scripting-Interface for the SIR-Library
  Developing RESpecCT in PyQt seems to be very promising. There is an apparent additional effort though: SIR is currently only available in C++, so we would have to develop SIR for Python. Since a fast and easy scripting interface to SIR is very desirable anyway, we would kill two birds with one stone. The additional amount of work at the beginning would give us a scripting interface almost for free. Even if we would later decide to switch to C++ development we will still have the scripting interface.

5.3.2 A small example

In order to get a better impression how powerful QT is in combination with Python, we show a small example-script:

```
import sys, qt                                  # load modules
a = qt.QApplication(sys.argv)                   # instantiate QApplication
w = qt.QWidget()                                # instantiate QWidget
w.resize(200,120)                               # resize QWidget
quit = qt.QPushButton("Pushme",w)               # instantiate button
quit.move(62,40)                                # move the button
    # connect the signal "clicked" of the button with the method "quit" of the widget
w.connect(quit,qt.SIGNAL("clicked()"),a,qt.SLOT("quit()"))
a.setMainWidget(w)                              # w is mainwidget of the application
w.show()                                        # show w modal
a.exec_loop()                                   # start the application
```

Figure 4: The small example-application.

This script pops up a little window with a button (Figure 4). If the user clicks on the button the application exits. Although this is a quite minimal example, it shows how easy and intuitive the code can be read. For an advanced PyQt-programmer, the code is as readable as the comments in this example.

6 SIR Wrapper

6.1 What is SIR

SIR stands for *SpecC Internal Representation*. It is the common data-structure used to store all language-constructs written in SpecC. The idea is that all tools use this data-structure, so they can easily exchange information between each other.

A clear and consistent data-structure also makes it easier to the tools to perform the individual refinement steps. Since a great advantage of SpecC is the consistency in all levels of the design process as well as the ability to simulate at every stage, the underlying data-structure is of great importance.

SIR is the data-structure which has been developed by Rainer Dömer at the CECS, and it fulfills the requirements well. Since there are a lot of functions and methods in SIR which are not needed to be accessed externally or which should never be used if not internally, SIR is divided into two levels.

SIR Level 1 contains all classes, methods functions and variables which are strictly for internal use (see Figure 5). *SIR Level 2* consists of all classes and functions which should be used by tools. Level 2 allows to perform actions at a much higher abstraction-level (see Figure 6).

6.1.1 Example: SIR_Behavior

As it may seem obvious, there are classes strictly dedicated to Level 1, like e.g. SIR_Member. Someone who wants to write a tool using the SIR does not even have to know about this class. Most of the classes however have a Level 1 part as well as a Level 2 part. There are no such classes without a Level 1 interface.

SIR_Behavior is an example for such a class. It's definition is listed in Appendix C.1, it's interface-file is listed in Appendix C.2. As we can see, the two files are quite similar, however there are differences. How and why these changes were performed is explained in Section 6.4

6.2 SWIG

As RESpecCT covers the whole refinement-process it seems to be a very thorough test-application for such a data-structure. Remains the problem, that the SIR currently only exists as a C++-library. The decision was made - on the other hand - to develop RESpecCT in Python.
Further research on this problem reveals, that there are three possible solutions:

1. Rewrite SIR in Python.
   This approach would be very time consuming though very easy to integrate into the application afterwards. Although it would be possible to implement just the subset of the SIR currently needed in the GUI, we do not go for this solution because you would have to maintain both versions of SIR and make sure they really produce the same SpecC-Code.

2. Wrap all the classes methods and functions of SIR in Python-classes, -methods and -functions.
   Doing this has the advantage, that there is only one SIR, so if some internals change the wrapper does not have to be updated (unless the API is not affected). But still, this would involve a lot of work, probably more than rewriting it completely and it would be difficult to build a clean and consistent interface like that.

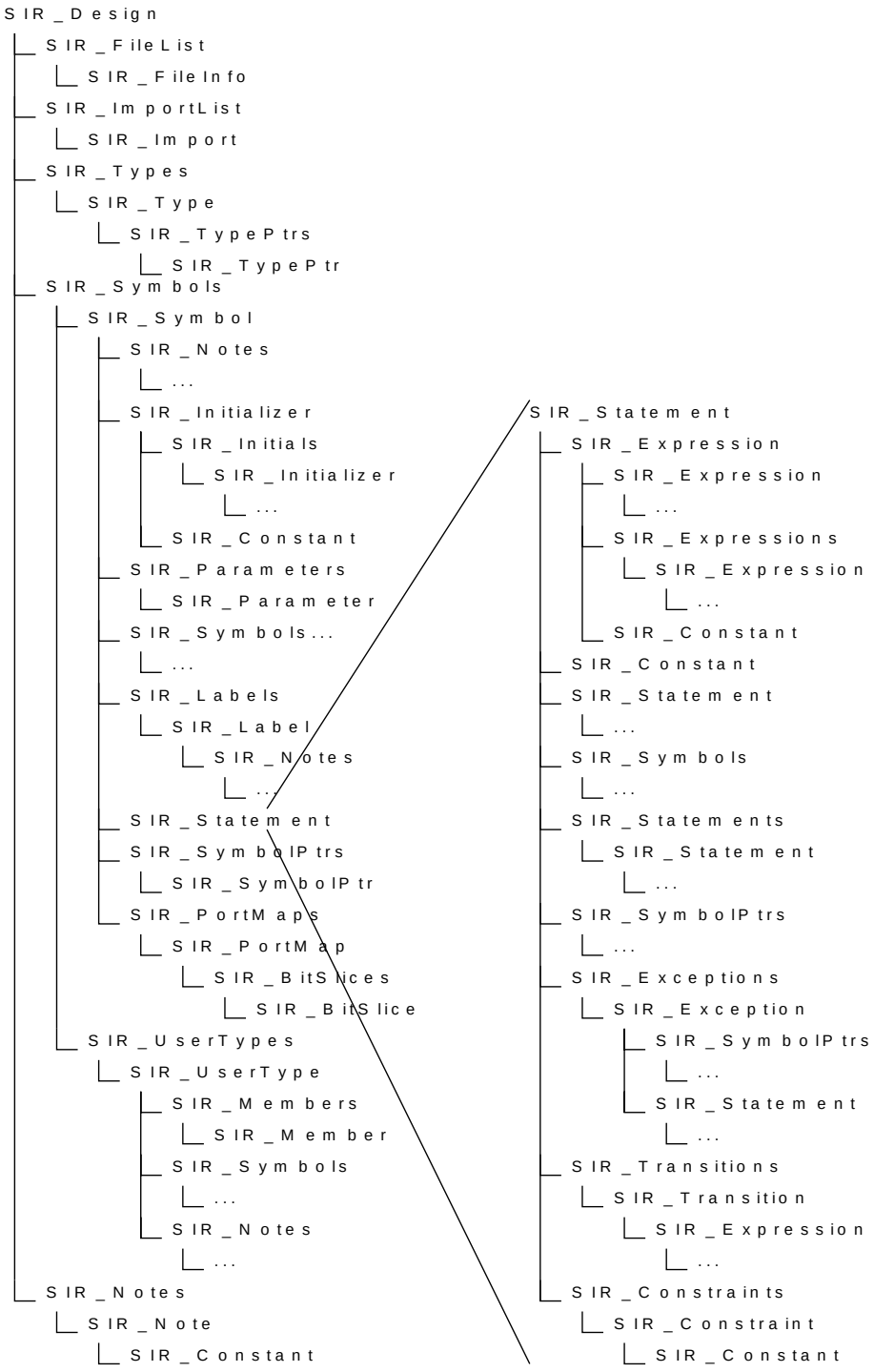


Figure 5: SIR Level 1 [3]

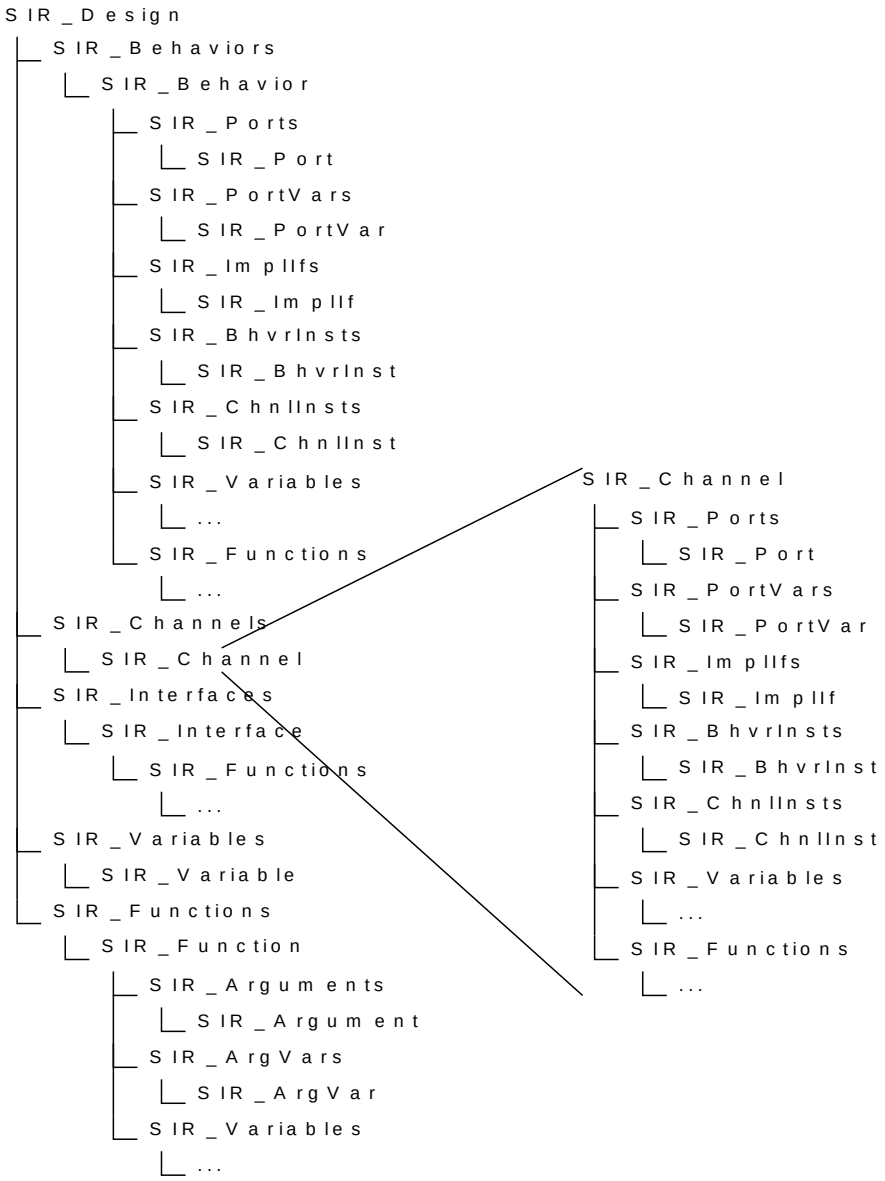


Figure 6: SIR Level 2 [3]

3. Use a tool like SWIG to automate the generation of the wrapper as much as possible.
   Since this is the approach with the least programming effort involved, it is the most flexible one and the easiest to maintain. Though the interface-generator cannot reflect the library perfectly, it should be possible to do special adjustments by hand.

Therefore SWIG was the choice. It is a quite advanced tool and there are over 400 pages of documentation on the web. The general workflow is the following:

1. Write an interface-file
2. Run SWIG on the interface-file and generate a C wrapper-file and a python shadow-wrapper
3. Compile the wrapper-file to an object-file (e.g. with gcc)
4. Link the generated object-file against the SIR-library into a shared library

6.3 Creating SWIG-Interface-files

Creating the interface-files is the first and most important task using SWIG. The common procedure is to make a copy of the header-file of the class one wants to wrap and perform certain modifications on it. In this header-file one has to specify the name of the module e.g.:

```
% module definition
```

and the names files one wants to include (since SWIG does not follow normal ”#include”-statements) plus the name of the original header-file:

```
%{
#include "Global.h"
#include "IntRep/Symbol.h"
#include "IntRep/ Definition .h"
%}
```

Now we have created a correct module-file. If we save it as ”modulename.i”, we can run SWIG on it. Usually, however, one does not stop here. Since we don’t want SIR Level 1 - methods to be used, there is no point in making them available in Python through the wrapper. The next step will be to remove all Level 1-methods. In order to put everything together, after we have created all interface-files, we create another one called sir.i which includes other interface-files. Like this, there will be only one Python-module produced, containing the whole Level 2 SIR.

6.4 Modifications and problems

This is the theory. This is how one wishes it to be - and perhaps in a couple of years we come to that point. Reality, however is different. We knew there were still some constructs in C++ SWIG can not handle. Some are likely to be solved in the near future, some are of a more general nature. Templates are one mechanism SWIG can not handle. Especially template-classes. With this we had to deal somehow, since most of the classes in SWIG are inherited either from SIR_List or SIR_ListItem which both are template-classes.

6.4.1 Templates

This issue - though annoying - revealed itself solvable since we don't create new types of SIR_List and SIR_ListItem. The only thing we would have to do, is to remove the inheritance-statement in the affected classes and insert method-prototypes which otherwise would be extracted from the template-class. For SIR_Behaviors the declaration in the header-file looks like this:

```
class SIR_Behaviors :                /* behavior classes list */
    public SIR_List<SIR_Behavior> /*is inherited from SIR_List (Template-class) */
{
public:
 (...)
```

In the interface-file the inheritance is removed and replaced with method-prototypes:

```
class SIR_Behaviors                  /* behavior classes list */
{
public:

bool Empty(void);                    /* test for empty list ? */
unsigned int NumElements(void);      /* number of list elements */
SIR_Behavior * First (void );        /* first element (NULL if empty) */
SIR_Behavior *Last(void );           /* last element (NULL if empty) */
SIR_Behavior *Previous(void );       /* previous element (NULL if none) */
SIR_Behavior *Curr(void );           /* current element (NULL if none) */
SIR_Behavior *Next(void );           /* next element (NULL if none) */
SIR_Behavior *Prepend(SIR_Behavior *Elem);
SIR_Behavior *Append(SIR_Behavior *Elem);
SIR_Behavior * InsertBefore (SIR_Behavior *Elem,SIR\_Behavior *Succ);
SIR_Behavior * InsertAfter (SIR_Behavior *Elem,SIR\_Behavior *Pred);
SIR_Behavior *Remove(SIR_Behavior *Elem);
SIR_Behaviors *Concat(SIR_Behaviors *Appendix);
SIR_Behaviors * Precat (SIR_Behaviors *Prependix);
 (...)
```

The template will now not be visible any more from Python, but since the Python-module will be linked against the SIR-library, the template is called implicitly because in the C++-compiler does exactly the same as we did manually.

This issue successfully solved, there still remains a lot work to be done in modifying all those interface-files. Since the procedure is very straightforward, we decided to automate this process. We wrote then a Python-script which takes a header-file, inserts the ”module” and ”import”-statements, examines it for the template-classes and inserts the method-prototypes of the appropriate types. This effort turned out to be very worthwhile since a change in the header files just requires a regeneration of the interface-files in order to be correctly reflected in the python-wrapper. A very simple change in the header of the template-classes could otherwise require some hours of editing the interface-files. The Python-script for the automatic interface-generation in included in the appendix (Appendix C.3)

6.4.2 Typedefs

Another problem are typedefs. SWIG does understand typedefs, but is does not reflect them entirely. In the SIR-library there are often typedefs used like ”typedef class SIR_Behavior sir_behavior”. Then there are methods returning values like: ”sir_behavior *Copy(const char *Name, BOOL Strip=False);”. Python will not create a wrapped object around the returned pointer and will throw a TypeError or AttributeError using it. This can be solved, by changing the method-prototype to ”SIR_Behavior *Copy(const char *Name, BOOL Strip=False);”. In this case, Python will recognize the returned value as a pointer to a SIR_Behavior and wrap it correctly. Now this is fixed very fast, but there are a lot of methods where this problem occurs, so again it is a lot of work changing it manually. So far we could not automate it though, since it is not a trivial adaption. The class ”SIR_PortMap” e.g. has a capital letter inside, so there is no simple rule to perform the transformation from ”sir_portmap”. Perhaps we will consider to make these changes directly in the SIR header-files, since the changes would not break any other code.

6.4.3 Pointer to Pointer

After considering all of this, one can use the Python-wrapper and it works well. Only while using it, one will encounter further issues which require some attention. In C and C++ a method or function only can return one value. If one wants to return several one can think of introducing an intermediate STRUCT holding all of the return-values and return it. This can sometimes make sense, but can also produce confusing code. The other possibility is to return supplementary values through arguments. The arguments have to be pointer to a pointer. The called method assigns them a value and like this the caller can use the value by dereferencing his argument. This has the advantage to produce code which looks simpler, but depending on the situation it can also be more difficult to read since the information-flow is not obvious any more. In SIR the latter method is used deliberately. Python has no pointers, only references. The conversion between references and pointers is done implicitly. Therefore, creating a pointer to a pointer is not possible. This makes it impossible to call such methods directly from within Python.

The solution is to add methods to the C++-interface-files which take no pointers to pointers as arguments. These methods call the corresponding methods and return whatever value one wants. It is obvious, that this can not be done easily in an automated way - and this means work. Also it has to be adapted every time the interface changes. In addition, since there is only one return-value, sometimes there have to be added several functions in order to wrap one method with more than one argument. This is the approach we took so far. It has the huge disadvantage, that we lose consistency between the two languages. Before, the documentation for the SIR-library had perfectly covered the Python-wrapper. Since we have introduced supplementary functions, this is no longer the case. The changes have to be documented separately.

6.4.4 Function Overloading

Function and method-overloading is a very common and convenient concept in C++. However there is no correspondent for this concept in python. SWIG handles this problem by ignoring all multiple declared functions and methods except the first one encountered. This is a very simple way and apparently it provoke missing functionality. One way in making methods available to python is to add methods with a different name in the C wrapper-code which call the overloaded functions/methods. This is very simple to perform (could be even done automatically) but results in inconsistencies with the C++-interface.

A way to avoid this is to add methods to the python-wrapper, which check the types (and number) of the arguments and call the (previously renamed) appropriate C++-methods. This is a bit more work, but

afterwards the interface stays the same. The Python method can be called like the overloaded C++-method.

In the current status we made only those methods available which are actually used. Only when it was really necessary we implemented C++-accessor methods. The approach with Python type-checking and the wrapper-method mentioned above is desirable to be used in a more advanced state of the Application.

6.4.5 Other Issues

Besides the problems discussed above, there still remain a couple issues. Some advanced C++-features like friend classes, nested classes, operator overloading and namespaces, SWIG is currently unable to translate. We did not have to worry about these since they are not exposed in the SIR-Interface.
An issue we actually have sometimes to deal with are exotic data-types like ”long long” or ”long double” and advanced typedefs like ”typedef ERROR (*sir_bhvr_fct)(sir_behavior*, void*);”. For problems like these we have to decide individually how to handle them.

6.5 Compilation

6.5.1 Unix

The compilation under Unix follows three basic steps:

1. Generate C-Wrapper with SWIG

   swig −c++ −shadow −python $(MODULE).i

   This runs SWIG on the interface file. The option ”c++” tells SWIG that is has to deal with C++ code, otherwise it assumes there is C code. ”-python means that we want to generate a python module. ”-shadow” means that SWIG should generate python shadow classes which correspond to the C++-classes. If this option is omitted, the python interface is ”flat”, without hierarchy. In this case we would have several hundred static functions which operate the underlying C++-objects. To have shadow-classes is very convenient, since the library can be used in the same manner as it would be used from C++.

2. Compile the Wrapper

   c++ −fpic −fpermissive −c $(MODULE)_wrap.c $(INCLUDE_PATH)

   When compiling the wrapper into an object we have to make sure, that all original SIR header files are in the include path. The option ”-fpic” is needed since we want to create a shared library. It makes the compiler produce position-independent code. The option ”-fpermissive” is needed with newer versions of gcc since SWIG generates apparently not 100% ANSII-C conform code. This option causes the compiler to be less restrictive and throw warnings instead of errors for certain constructs.

3. Link it into a Shared Library

   c++ −shared −fpic $(MODULE)_wrap.o $(LIBS) −o $(MODULE)cmodule.so

   Linking into a shared library can be painful, but with the right compiler-version (we use gcc.2.95.2) and the right options it usually works. The flag ”-shared” signals the compiler to produce a shared library. We have to include our just created object, the compiled SIR-library (position independent) and the python library.

6.5.2 Windows

In the windows-environment, shared libraries are called DLL's. Generating these is comparatively more complicated. It gets even more complicated because of the fact, that C++-SIR does currently not compile with Visual C++. Therefore we use gcc under cygwin. It makes no sense to explain the build-process in detail here, here a listing of the commands to get an impression of how it works:

```
swig -c++ -shadow -python sir.i

# compile the library
c++ -fpermissive -c sir_wrap.c $(INCLUDE_PATH)

# generate the dll:
gcc -s -Wl,--base-file,sir.base python20.dll sir_wrap.o $(SIROBJS) \
    -lstdc++ -mdll -o sirc.dll -Wl,-e,_sirc_init__FPvUlT0@12

# generate the .def-file
dlltool --base-file sir.base --output-def sir.def --export-all-symbols --dllname sirc.dll sir_wrap.o

# generate the .exp-file
dlltool --base-file sir.base --def sir.def --output-exp sir.exp --dllname sirc.dll

# regenerate the dll:
gcc -s -Wl,--base-file,sir.base sir.exp python20.dll sir_wrap.o $(SIROBJS) \
    -lstdc++ -mdll -o sirc.dll -Wl,-e,_sirc_init__FPvUlT0@12

# repeat .exp-file-generation:
dlltool --base-file sir.base --def sir.def --output-exp sir.exp --dllname sirc.dll

# final dll-generation:
gcc -s sir.exp python20.dll sir_wrap.o $(SIROBJS) \
    -lstdc++ -mdll -o sirc.dll -Wl,-e,_sirc_init__FPvUlT0@12
```

6.6 Summary

If we consider the amount of work and maintenance it would take to rewrite SIR in Python or to manually wrap it, SWIG is a very convenient and helpful tool. But there is no magic involved, so there are still quite a lot of steps to go through until one obtains a conveniently usable wrapper.

Once this is done we have SIR completely accessible through python with almost the same API than the C++-version – which is not only useful for RESpecCT but also represents a light, fast and platform-independent scripting-interface which can be used to implement small tools using SIR quickly.

If at some point it may seem desirable to have a perl or a Tcl/Tk -interface to SIR, SWIG can generate those from the same interface-files with only slight changes.

7 Implementation

This chapter will describe shortly the implementation of each element in the application. Is covers design decisions made and wants to give some background on user-interface design [4]. It is a quick overview of how RESpecCT is built and what structure is behind.

We will discuss some general concepts as well as specific classes and where they are derived from. In order to understand this fully one needs some knowledge of the QT-library which is intensely used. The QT-classes are not further described, please refer to the official QT-Documentation ([5] or [6]). All QT-Classes start with a capital ”Q” followed by a descriptive name like ”QWidget”.

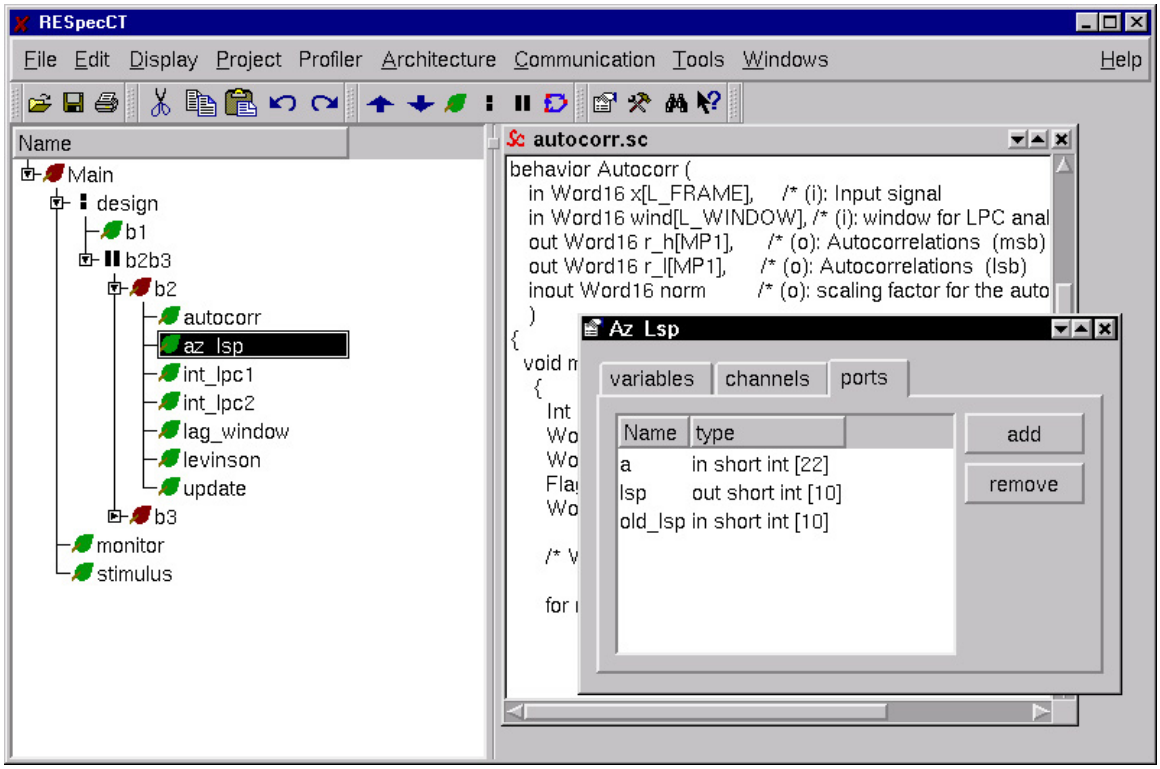


Figure 7: The RESpecCT Main Window

7.1 First Steps

Once we have a Python-version of SIR we can actually start implementing the actual application. So what do we need? And in which order?

The first element used is a Main-Window with a toolbar, a menu and a statusbar. Then we have to integrate

some tree-view to display the behavior-hierarchy. Once we got that tree-view, we can start using the SIR-module and continue expanding.

7.2 The Main Window

Figure 7 shows the RESpecCT main-window. It is based on a QMainWindow with a menubar, a statusbar and several detachable toolbars. All of those components can be hidden in order to maximize the workspace. To organize the workspace we put a horizontal QSplitter in the middle. It separates the workspace into two parts at a predefined position, this position can be moved by the user by simply dragging it to the left or to the right. The left side will be reserved for the behavior-tree, in the right side we put a QWorkspace. The QWorkspace can be used as an MDI Workspace which means that one can open a random number of child-windows inside the workspace. They can deliberately moved within the workspace, but always stay inside. The user can resize them, tile them (resize all windows so they fit together in the Workspace) and cascade them (resize all windows to the same size and put them one behind the other).

7.2.1 Behavior Tree

The behavior-tree class is called SC_tree. It inherits from QListView. It adds methods for filling itself from an SIR-Design, column management, behavior-mapping and some more. When filling itself from a SIR_Design it just creates the toplevel-item, the rest of the design is read recursively from within the items.

The class for the list-items is called SC_item. Obviously it is inherited from QListViewItem. It's constructor takes a SIR_Instance and creates his immediate children. Each item checks his own type and automatically uses the according icon. There are different icons for behaviors of type serial, parallel, leaf, FSM and other. Each SC_item keeps a reference to his SIR_Instance and SIR_Behavior. In addition to that there is a dictionary in SC_tree which holds key-value-pairs of the type "SC_item:SIR_Behavior". This may seem redundant information but is due to the current characteristics of PyQt. Every QT-Object is actually a C++ object. Every time the QT-code returns an instance to python, it is automatically wrapped into a Python wrapper-object. As long as we don't subclass the QT-object there is no problem, but if it is sub-classed, there is. Python's memory management automatically garbage-collects any object which is no longer referenced in the code. In the case of the tree, only the QListview would hold a pointer to its items, which is in C++. The python wrapper object would be deleted as soon as it gets out of scope in the Python-code. Any operation on the SC_tree which returns an item would then return a QListViewItem which is no longer wrapped in an SC_item, the Python wrapper-object, since that has been deleted.

To prevent this, we have to keep a reference to the python object (SC_item) in the application. Then the python-object is not deleted since its reference count is still positive. The SC_tree-operations will now return items of the sub-classed type since the wrapper-object persists.

Often it is convenient to keep references to all objects somewhere, but if one forgets it, there can occur very strange errors. After discussing this issue in the PyQt-mailinglist, we agreed that it would be convenient if PyQt kept references to the sub-classed objects transparently. Thus, in the next release of PyQt (version 2.3) there will be no longer a need to think of this.

7.2.2 MDI Workspace

MDI means Multi Document Interface. In modern desktop applications it is a common concept which allows the user to open multiple documents or - more generally speaking - child-windows and let him move them

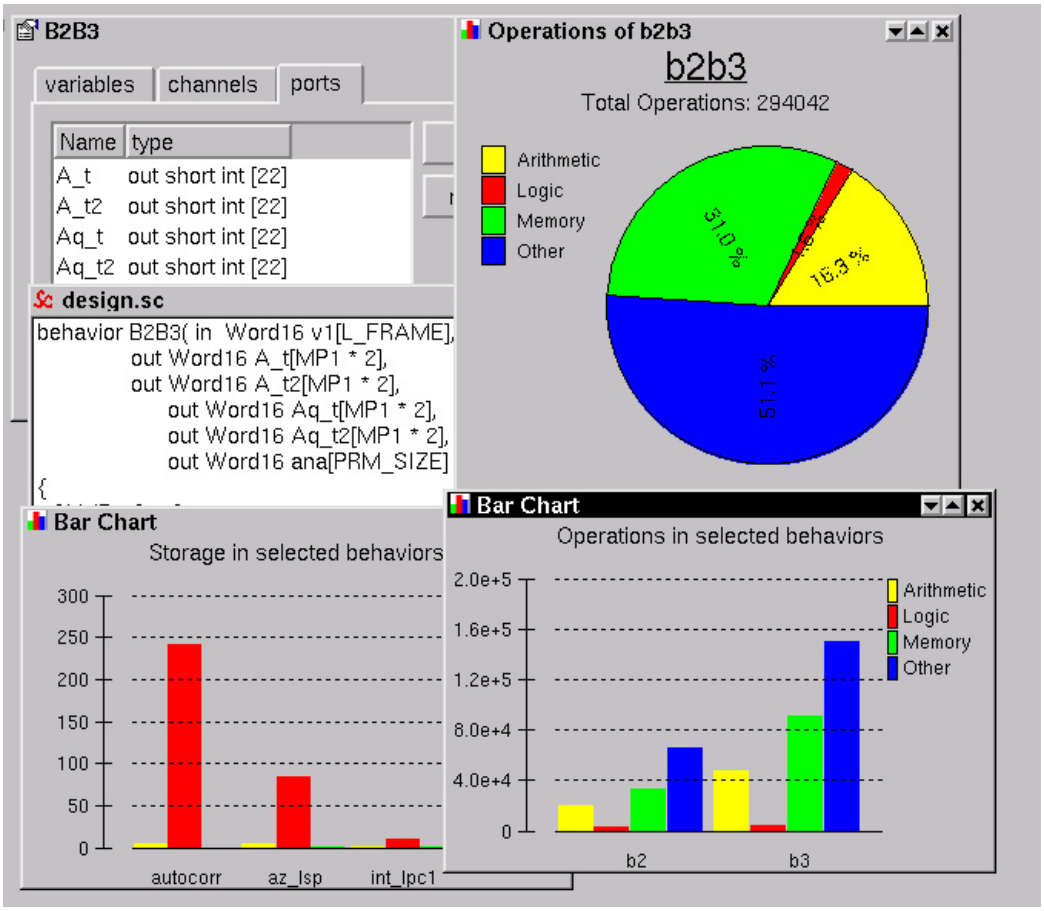


Figure 8: MDI Workspace

around deliberately while still keeping them in the main application-window. This is designed in order to let the user freedom in configuring and adapting the user-interface to his needs and habits, while preventing to let the GUI interfere too much with other applications.

The MDI-workspace of RESpecCT is shown in Figure 8. In the RESpecCT MDI-workspace one can do code editing, compare different profiling-charts, view and edit properties of behaviors and perform other refinement-steps at the same time. The danger with MDI is that one loses track of all open windows and that windows get hidden behind others. On the other if the user uses it wisely it can be a powerful tool. We think that the advantage in productivity is so big that it is worth taking the risk of losing track of the open windows.

7.3 Code Editor

RESpecCT - Refinement and Exploration tool for the SpecC Technology - is not an IDE. The name should make this clear. There are, on the other hand, several reasons which make a code-editor essential:

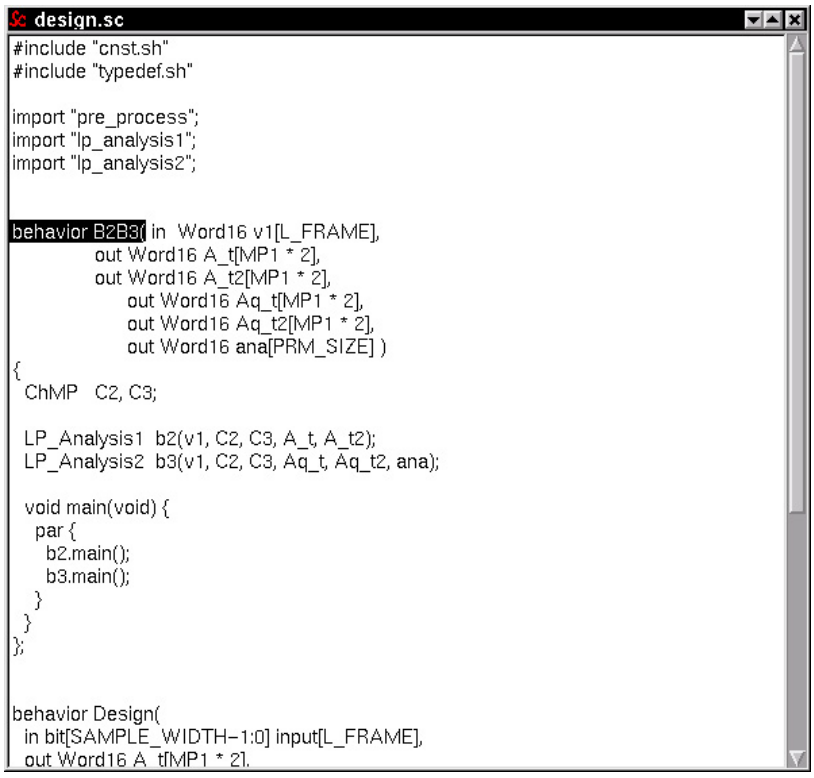


Figure 9: Code Editor.

- During refinement it is often necessary to edit the code and make small modifications.
- A more advanced SpecC-user will be faster to make certain changes directly in the code than rather using the user-interface.
- Not all modifications of the source code can be performed through the user interface. FSM-behaviors e.g. have to be treated very individually.
- For educative purposes or for testing it can be instructive to take a look at the source code in order to understand how the code evolves.

For all these reasons it was important to include a code editor from the start.
The editor integrated right now (Figure 9) is very basic. It is a QMultilineEdit-widget with some small enhancements for loading, saving and cut & paste. For a later version it is desirable to have features like syntax highlighting, unlimited undo and redo and transparent co-editing with the user-interface. This will however consume a considerable amount of work and a lots of issues concerning this are not clear yet. Thus we decided to go for a simple editor for the start and enhance it as soon as the rest of the application is more complete.

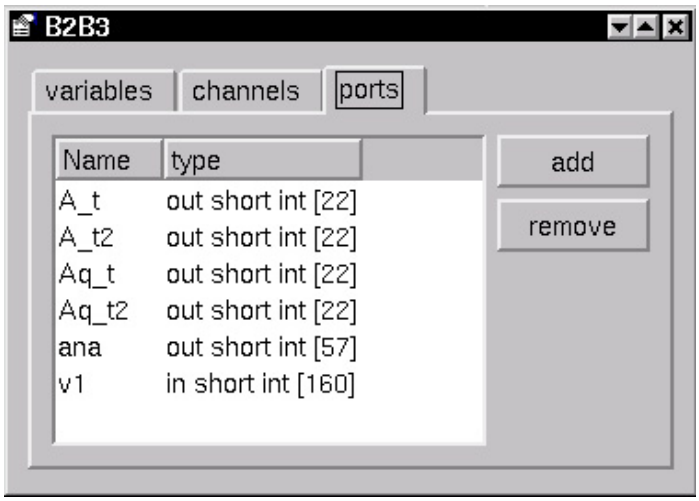


Figure 10: Properties Dialog.

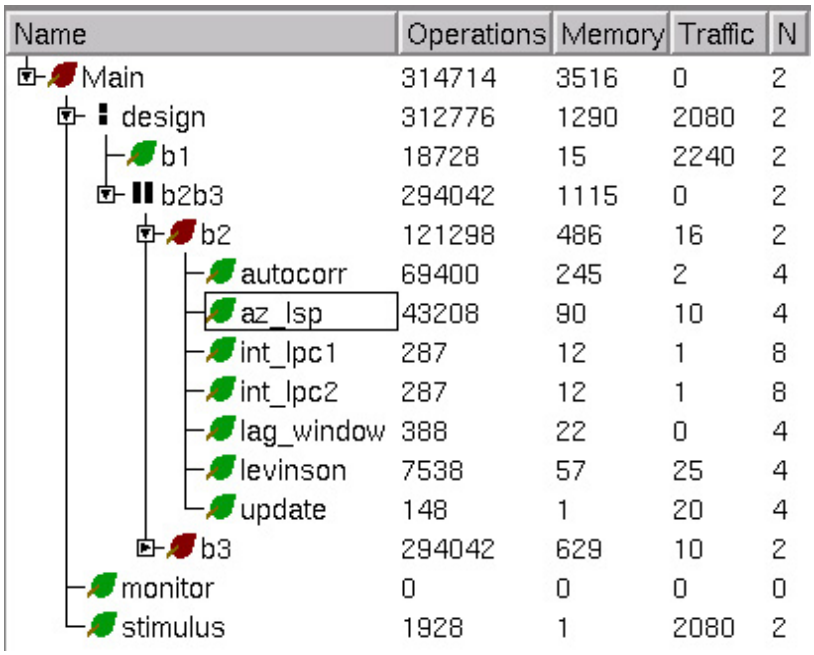
7.4 Properties Dialog

The behavior-tree is very good to visualize the hierarchy, to get a quick impression of the topology of the design and to check the type of the behaviors. It is not appropriate though as a display for the contents of the behaviors.

To display variables, channels and ports we designed the *editpr*-widget (Figure 10). It has a QTabWidget with tree tabs named variables, channels and ports. Each tab looks exactly the same. They contain a QListView with two columns, ”Name” and ”Type”. on the right hand side there are two QPushbuttons, ”Add” and ”Remove”. The properties-dialog is accessible either through the popupmenu of the behavior-tree or through an icon in the application-toolbar. The dialog scales nicely on resizing and can be minimized or maximized with the usual buttons in the upper right corner. The design of this dialog has been made using QT-Designer. The generated class is called ”editpr”. The inherited class is called ”bhEditProp”. It provides the editor with the required functionality. It takes an SIR_Instance as parameter, and fills his QListViews with the properties of this behavior. The title-bar of the dialog will display the name of the behavior. The buttons add and remove are meant to let the user add and remove properties. The remove button currently shows the user dependencies and leads him to the appropriate line in the SpecC-code in order that he can decide himself what to do. The add-methods are currently not implemented. They should start a wizard which leads the user through the process of adding a new property.

7.5 Profiler

The general task of the profiling is to compute all kinds of informations about the design. These informations should help the user to make design decisions like allocation and partitioning. The profiling-tool itself is a command-line-tool currently developed by Lucai Cai. It takes a pointer to a SIR_Design. This design will be annotated with some information. In oder to interact with the user-interface and to integrate the tool more closely, we decided to specify an API for the profiling-tool and to compile it into a shared library. like this



Name	Operations	Memory	Traffic	N
Main	314714	3516	0	2
design	312776	1290	2080	2
b1	18728	15	2240	2
b2b3	294042	1115	0	2
b2	121298	486	16	2
autocorr	69400	245	2	4
az_lsp	43208	90	10	4
int_lpc1	287	12	1	8
int_lpc2	287	12	1	8
lag_window	388	22	0	4
levinson	7538	57	25	4
update	148	1	20	4
b3	294042	629	10	2
monitor	0	0	0	0
stimulus	1928	1	2080	2

Figure 11: Columns with Profiling Information

we could wrap it into a python module and use it interactively.

7.5.1 Columns

In order to evaluate profiling-results, the user wants to have a display which allows him to view as much information as possible as fast as possible. Thus it seems very convenient to put the profiling results as additional columns in the behavior-tree (Figure 11). Like this the user can choose the portion of the behavior-tree which is most important to him by opening and closing branches and has the corresponding data right besides it.

After profiling we display some columns which we consider as the most interesting ones. Depending on the design characteristics there may be special data which is of great importance to the designer. Therefore he is able to choose the columns which he wants to display from a big variety.

The columns displayed by default are *Operations, Traffic and Memory Consumption*. There can be chosen columns for all different kinds of operations (like addition). For every operation there can be columns for every data-type (e.g. integer 16 bit). Memory consumption and traffic can also be examined for every data-type. Since there is 29 different data-types and 56 different operations in SpecC, there will be about 2000 different possible columns to choose from. In addition there are some higher level metrics computed from others.

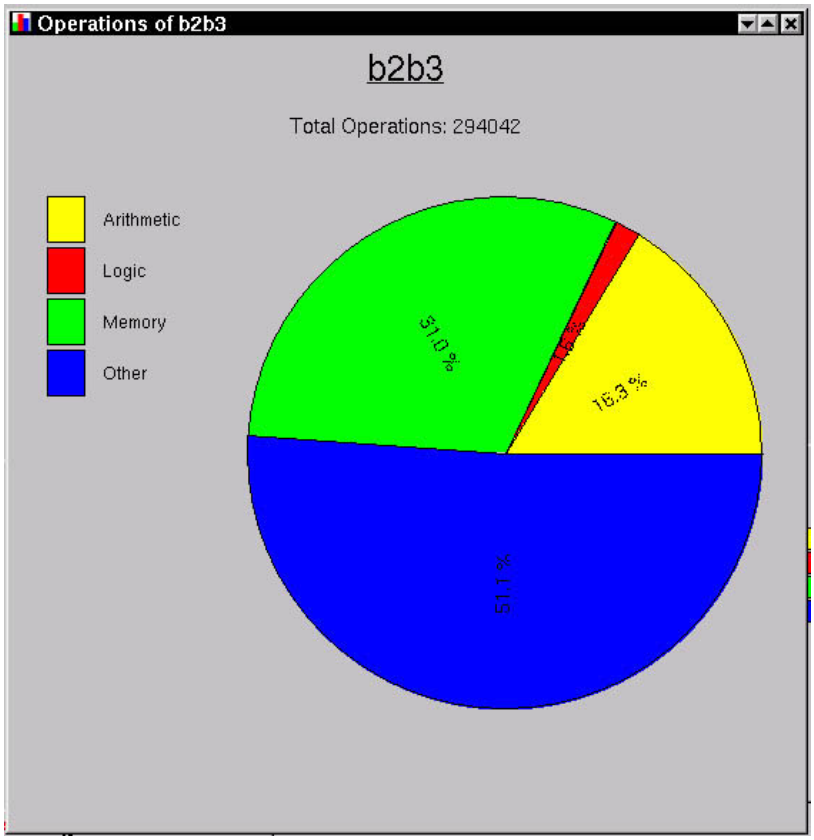


Figure 12: Pie-chart Widget.

7.5.2 Pie Chart

A disadvantage of the profiling-information in the columns is, that you only see numbers. Though most people today are quite habile in comparing numbers, it is much more intuitive to compare information with a more graphical representation. Therefore we decided to implement charts.

A pie-chart is a easily readable chart. Our Pie-widget (Figure 12) can take an arbitrary number of key-value pairs and displays them as different colored slices in the pie. It has a legend which can be aligned left, right or hidden. The overall sum is displayed in the subtitle, the amount of every single slice can be displayed in the slice itself (either as absolute value, or as percentage).

The pie-chart-widget is used to display one criteria for one behavior (e.g. different kinds of operations for behavior ”b2b3”).

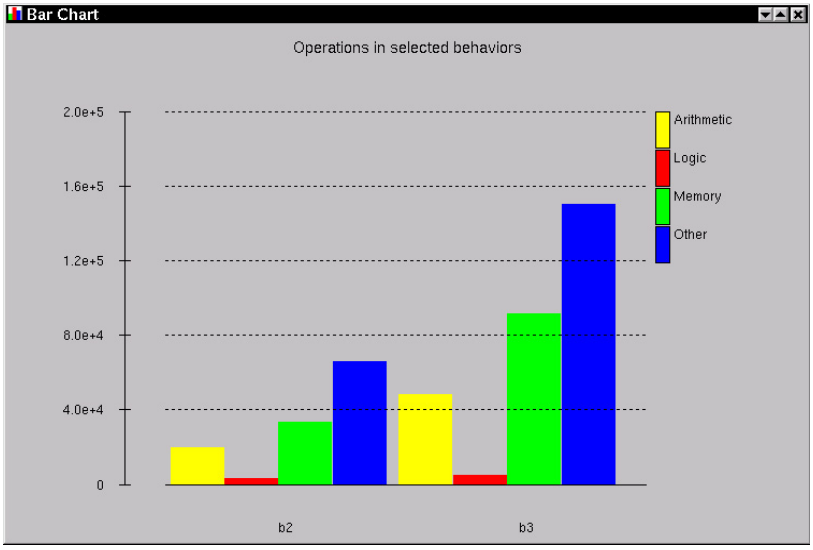


Figure 13: Bar-chart Widget.

7.5.3 Bar Chart

Sometimes it is important to compare different behaviors for the same criteria. We could open several pie charts and put them side by side, but this is not very satisfying. So we implemented the barchart-widget (Figure 13). The barchart can display one criteria for several behaviors. It scales automatically to whatever data it contains, the caption of the scale changes depending of the numbers to scientific or fixed display. If there is negative data, the 0-line is moved to wherever it fit's best.

Using the "shift" or "ctrl" – button, the user can select different behaviors form anywhere in the behavior-tree. Once he chooses the criteria to be displayed the bar-chart will be shown in the MDI-workspace. It has to be mentioned, that depending on the size of the screen it may not be advisable to select more than 10-15 behaviors at once.

7.6 Architecture Refinement Tool

After profiling is done, we can use the information obtained to make architecture refinement decisions. Architecture refinement consists basically of allocation, partitioning and scheduling. We had to find ways to make these decisions quick and easy to take.

7.6.1 Allocation Dialog

Allocation is basically the process of selecting the processors one wants to use in the design out of a list of processors offered from an IP-database. The display we developed is shown in Figure 14.
The allocation-dialog contains two QListViews. On the left there is the list of available processors, on the right the list of allocated processors. Between them there are two buttons to add or remove processors to the design. On the very right, there are standard-buttons with captions "OK", "Cancel", "Rename" and "Help".

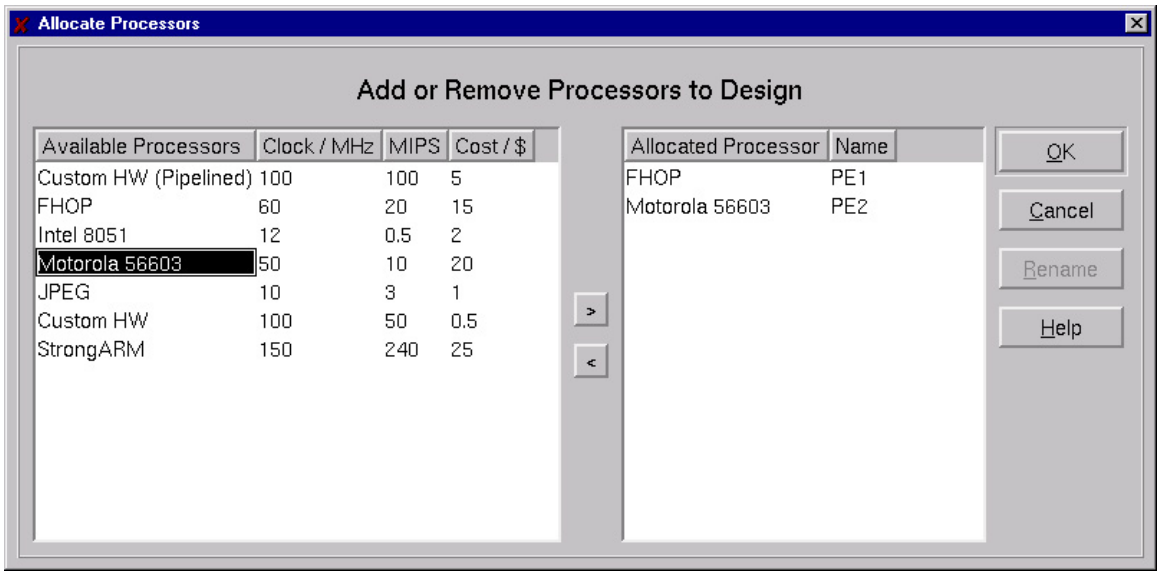


Figure 14: Processor Allocation.

The list ”available” shows currently just an internally stored list of processors with information like clock, performance and cost. This information should be taken from an IP-database later on. If you push the add-button, a dialog will pop up, asking you for a name (Figure 15) - since it is perfectly allowed to allocate several processors of the same kind, you have to give them a name in order to be able to identify them.
The application will annotate the design with the information shown in the ”allocated”-list (key values pairs of processor-type and name). This information will be needed from the architecture refinement tool developed by Junyu Peng.

7.6.2 Behavior Mapping

After processors have been allocated for the design, there will appear an additional column in the behavior-tree call ”PE”. There is also added a menu-item in the popup-menu of the behavior-tree called ”map”. Clicking on it will show a sub-menu listing the names of all allocated processors. Choosing one of the processors will map the currently selected behavior to this processor (Figure 16). At the same time the design will be annotated with the appropriate information.
Behaviors which get not explicitly mapped will be mapped to the same processor as their parent-behavior. Once the behavior-mapping is finished, you can run the architecture refinement (select ”refine” in the architecture-menu). This will launch the first step of the architecture refinement. There will be an additional level of hierarchy introduced representing the processors. Also there will be additional behaviors responsible for the synchronization of the communication between the behaviors.

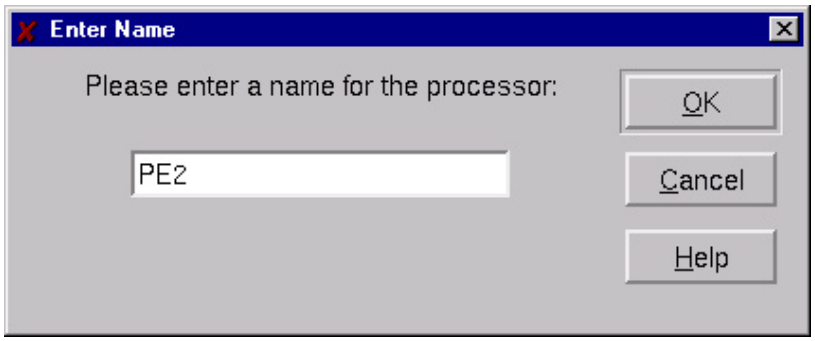


Figure 15: Nameenter Dialog.

7.6.3 Scheduling

After that there should follow the second step of the architecture refinement, the scheduling. Scheduling means to sort all behaviors on one processor in a certain order. There is currently no display for the scheduling. It is not just a one-dimensional problem since it should also be possible to schedule over the hierarchies. After the order of the execution has been selected by the user, the architecture refinement tool has to be run again in order to perform the changes in the SIR-datastructure.

7.7 Communication Refinement Tool

The communication refinement tool is currently developed by Samar Abdi. It's main task is to map the top-level-channels to busses and introduce transducers if necessary. Transducers are elements capable of connecting two components with different protocols.

7.7.1 Bus Allocation

Similar to the processor allocation there have to be allocated busses. Since this process is almost the same, we use the same display. This allows the user getting used to the GUI more quickly. As you see in Figure 17, there is on the left a list of available busses which lists a selection of busses with some data: descriptive name, Number of Address bits, number of data-bits and throughput in Mbit per second.
As with the processor-allocation dialog, one will be prompted for a name for the bus-instance (Figure 15).

7.7.2 Channel Mapping

After the bus-allocation one can select the item "map channels"in the menu "communication". This will open a dialog very similar to the previous (Figure 18). In fact, we used the "allocation"-dialog, and just applied a couple of modifications. In the list on the right hand side of the dialog figure all the top-level channels which have to be mapped to a bus before the communication refinement-tool can be run. In the QListView on the right hand side are listed all allocated busses. After selecting a bus and a channel, you can add that channel to the bus. It will be listed in a tree under the bus.

After all channels are mapped, the list on the right side is empty and the channel-mapping is completed. Now the dialog can be closed and the communication-refinement tool can be run by choosing the item

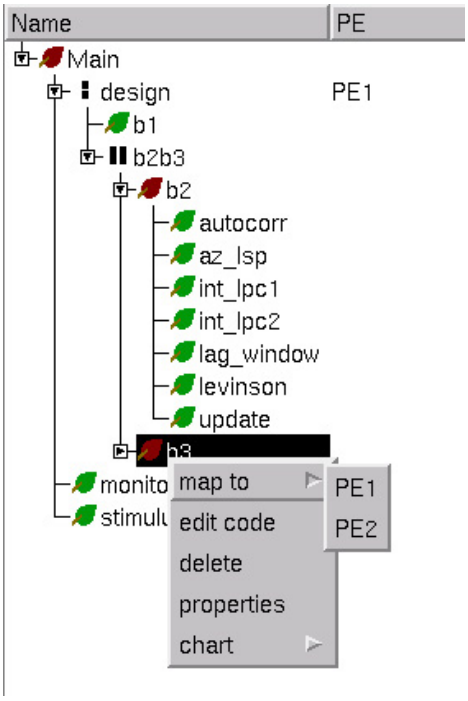


Figure 16: Behavior Mapping.

”Refine” in the menu ”Communication”.

Communication refinement will insert the appropriate busses and protocols into the design. If necessary there will be transducers inserted. A transducer will appear as an additional component on the top-level of the design.

7.8 Summary

Overall the implementation so far proves that python in conjunction with QT were a good choice for RESpecCT and enabled rapid application development. RESpecCT is now a basic framework showing all necessary concepts like accessing and modifying the SIR, integrating external tools and modules and provides basic widgets for data-evaluation and for taking decisions. This is what was fixed in the specification of the project (Section 4).
In addition to that it shows allocation and partitioning and can already be used to perform refinement to the architecture level on basic designs. Besides the GUI there is still work to be done in the tools. So many of the functions will only be available in the GUI after they have been implemented in the tools, but integration and extension is straightforward.

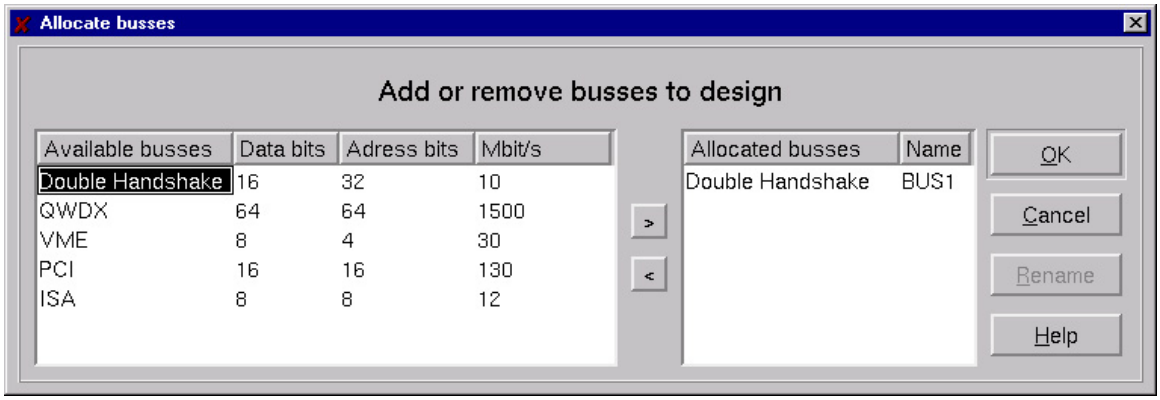


Figure 17: Allocation of Busses.

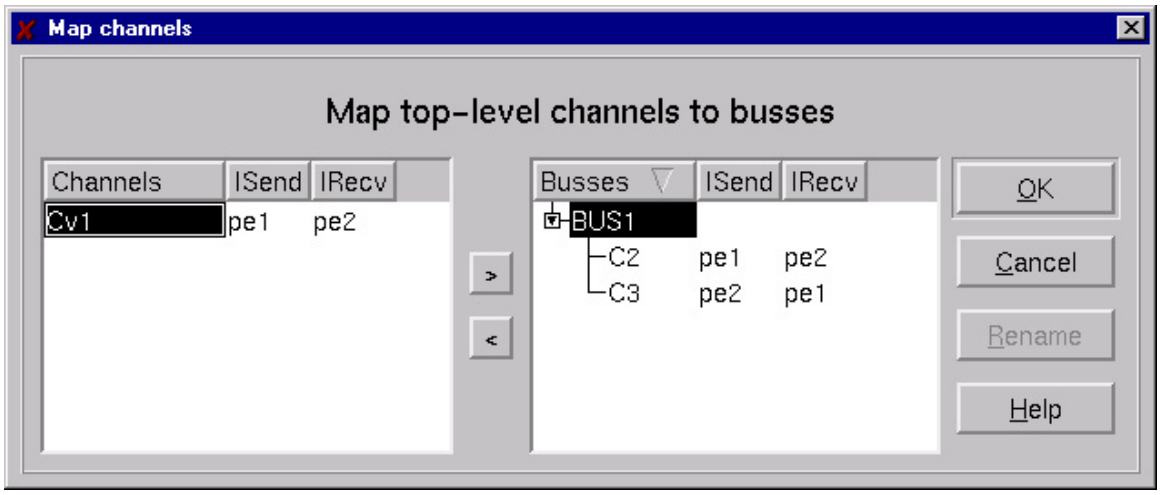


Figure 18: Map the Top-level Channels to Busses.

8 Example

Now that it is clear what was implemented and how, we want to show how the program actually works. In this section we describe the use of the program with the help of a small example-design.

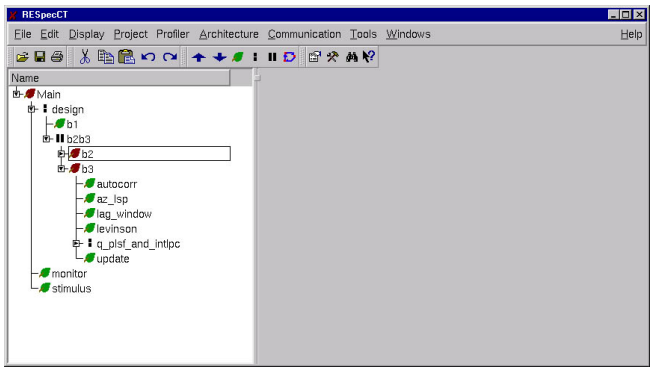


Figure 19: Load an Example Design.

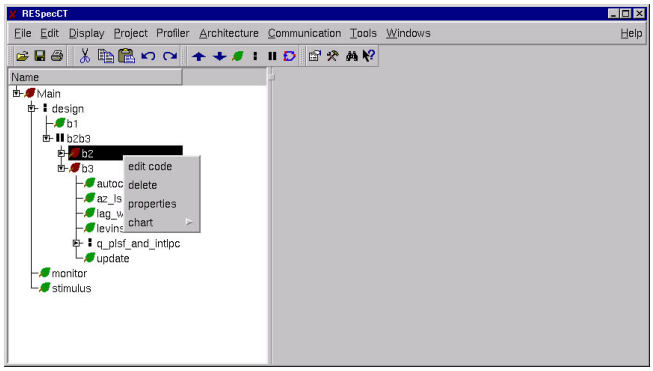


Figure 20: The Context Menu.

8.1 Loading and Examining Design

First we load a design into the project. We can do this by choosing the in the menu ”File” the item ”Open Design” or by just clicking on the open-icon in the toolbar. We choose our example-design called ”testbench.sir”. Instantly the behavior-tree of the design is displayed on the right hand side (Figure 19). Now the user can browse through the hierarchy, the symbols of the behaviors tell what type they are (serial, parallel, FSM, leaf or other).

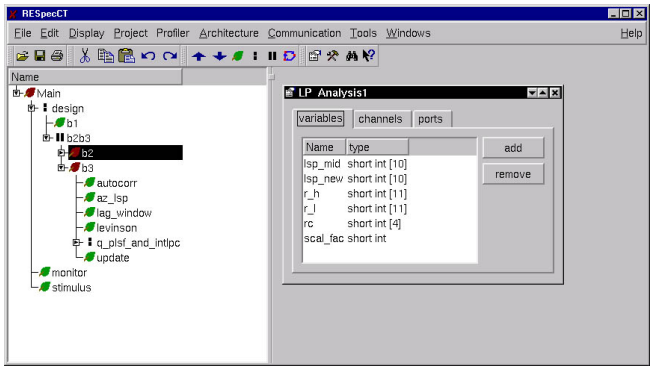


Figure 21: Variables of the Behavior.

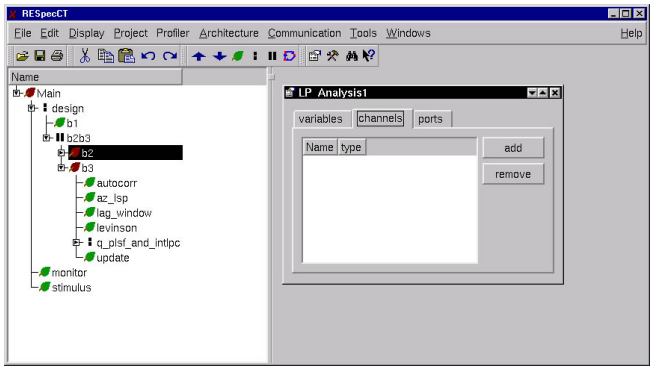


Figure 22: Channels of the Behavior.

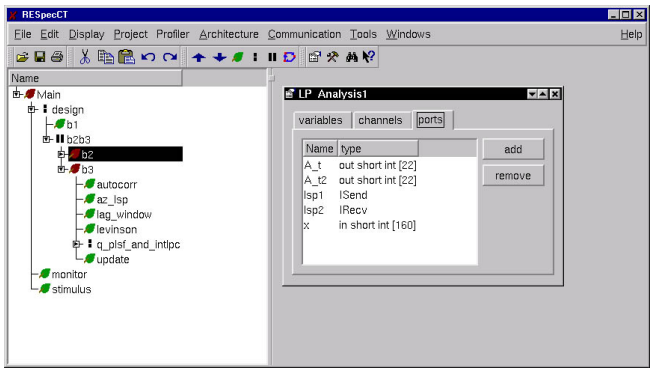


Figure 23: Ports of the Behavior.

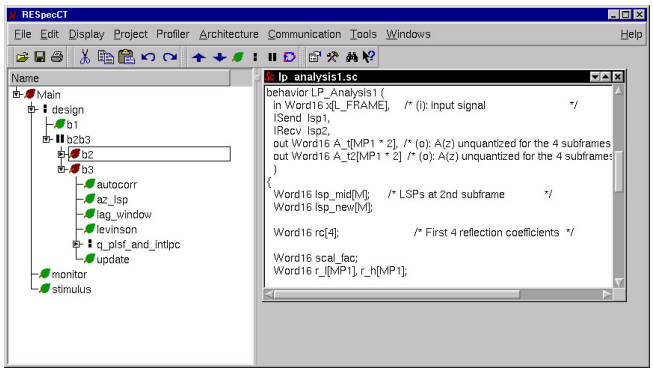


Figure 24: Source-code Editor for the Behavior.

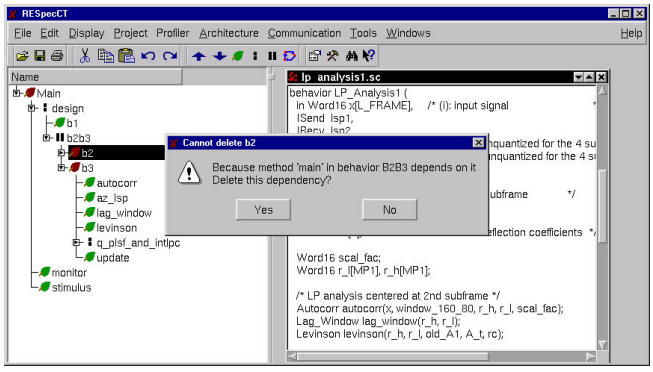


Figure 25: Evaluating Dependencies while Deleting a Behavior.

A right mouseclick on a behavior displays the context-menu (Figure 20). There the user can choose to view and edit the contained variables (Figure 21), channels (Figure 22) and ports (Figure 23), view and edit the source-code (Figure 24) or to delete the selected behavior (Figure 25).

8.2 Profiling

The next step is to run the profiling-tool. This is done by choosing the item ”Profile Design” in the menu ”Profiler”. This takes typically a minute. After this we get three additional columns in the behavior-tree. They display overall operations, traffic and storage for the behaviors. If the user wants to visualize the profiling results, he can choose one or several behaviors by clicking on them with the ”Control”-key pressed. In the contextmenu he then can choose the submenu ”Chart”. Now he can e.g. visualize the amounts of different types of operations (Figure 26). Depending of the nature of the data the user wants to visualize, bar-charts or a pie-charts are displayed.

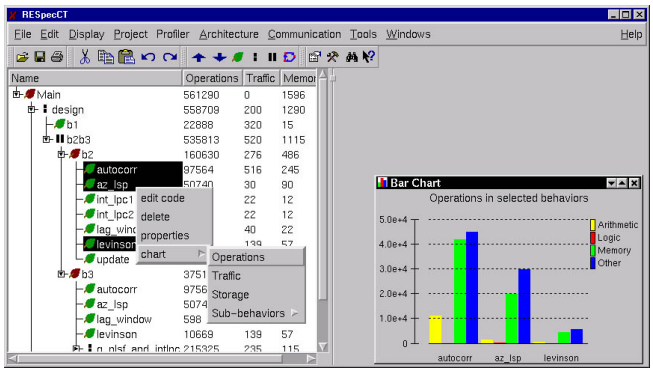


Figure 26: View Profiling Results.

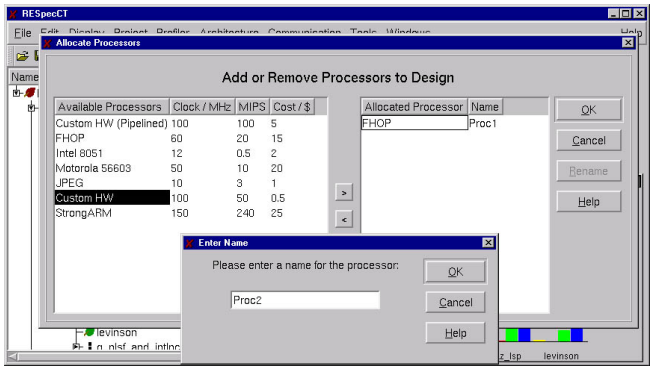


Figure 27: Allocating Processors for the Design.

8.3 Architecture Exploration

The main purpose of the profiling-information is to facilitate the decision-taking for the architecture-exploration. The first decision to take there is to choose the processors we want to use in the design. This process is called allocation of components (Figure 27). The next decision consists of mapping parts of the design to the selected processors. We call this partitioning (Figure 28). If a behavior is not mapped explicitly to a component, it will be mapped to the processor its parent is mapped to. After the mapping is performed the architecture refinement tool can be run. Basically the tool adds a new level of hierarchy to the design, which represents the processors and some additional behaviors, synchronizing the communication between the processors. The resulting design is shown in Figure 29.

8.4 Communication Refinement

As the communication refinement basically consists of introducing protocols in order to get the communication between components cycle-accurate, we first allocate busses and then map the toplevel channels to these

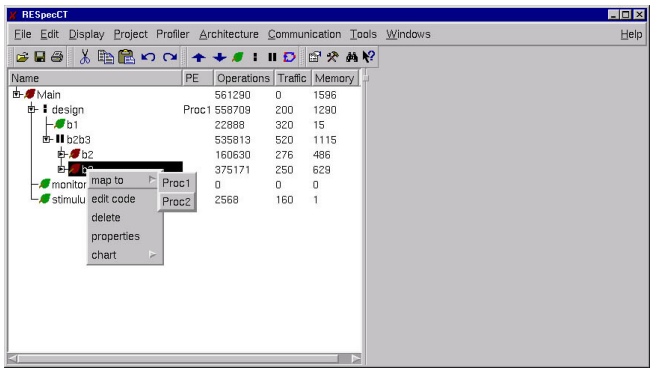


Figure 28: Mapping Behaviors to Processors.

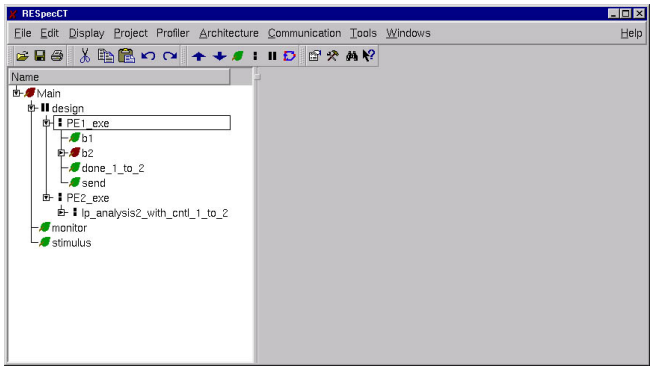


Figure 29: The Architecture Refinement Tool Introduces an Additional Level of Hierarchy.

(Figure 30). The bus-allocation can be called with the item ”Allocate busses ...” in the menu ”Communication”. As with the allocation of processors, we instantiate items of a list and give them unique names. Then we choose the item ”Map channels ...” of the same menu which will show the toplevel channels and lets the user map them to the allocated busses (Figure 31). If all toplevel channels have been mapped, the communication refinement tool can be called (item ”Refine” in the menu ”Communication”). The communication refinement tool will inline bus-protocols in the design. If the protocols of the components do not match to the connected bus, it will add additional components, called transducers which will transform one protocol to another (Figure 32).

8.5 Refinement to RTL

The refinement of the register transfer level is still in early research stages and therefore there is no display yet for it. We will develop displays and interfaces to this step as soon as it will be clear what information will be exactly needed in order to get an efficient implementation.

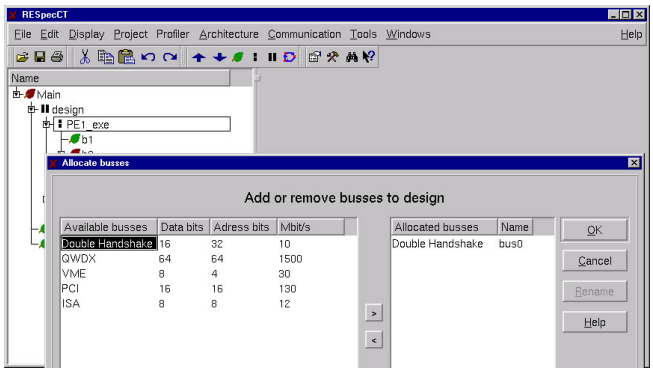


Figure 30: Allocating Busses for the Design.

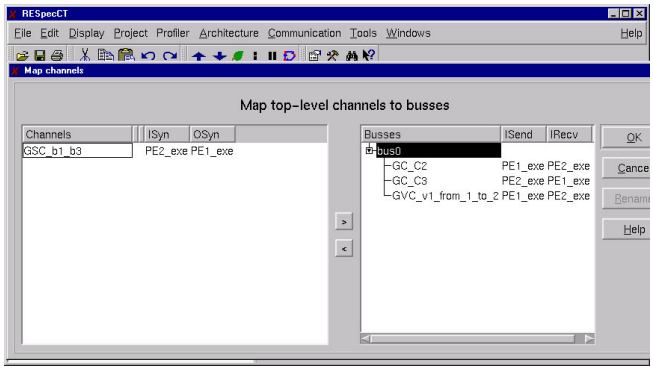


Figure 31: Mapping of the Toplevel Channels.

8.6 Summary

Although this is a very small example and therefore not all difficulties and special cases occur, it shows the basic refinement steps quite clearly. We see, that the user has to make certain decisions manually, other parts are automated and their result will reflect the quality of the decision previously taken by the user. Like this he gets feedback and can approach a solution more compliant with the original constraints in an iterative process.

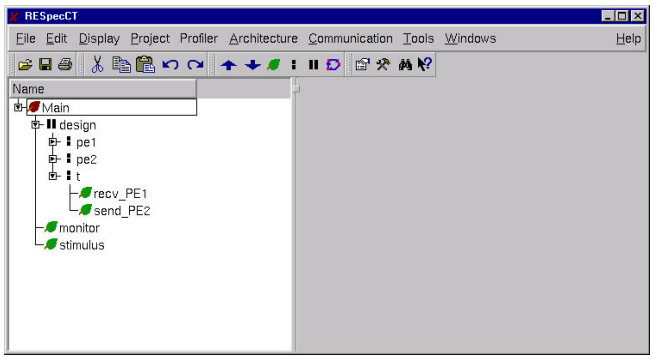


Figure 32: The Communication Refinement Tool Inserts Protocols and, if necessary, Transducers.

9 Conclusion

Although there has been done substantial work on this project, it is far from being finished. However, to finish it has never been the goal. The goal has been to establish a basic framework, create the essential widgets, come up with all basic concepts how things should be implemented and verify them.

To use Python as the programming language and QT as the toolkit was a keen try, but apparently it turned out to be the right choice. In very short time we managed to build a nice, friendly user-interface - and got a convenient scripting-interface for SIR almost for free.

RESpecCT is far from being an industry-standard EDA-tool, but it shows the principles and methodology of SpecC. Once it gets the lacking functionality it may convince people in the industry that SpecC is the Golden Way to go in System Level Synthesis. And perhaps in a couple of years SpecC marks the beginning of designs of so far unknown complexity and performance.

References

[1] D. Gajski, J. Zhu, R. Dömer, A. Gerstlauer, S. Zhao, *SpecC: Specification Language and Design Methodology*, Kluwer Academic Publishers, 2000.

[2] D. Gajski, J. Zhu, R. Dömer, A. Gerstlauer, S. Zhao, *The SpecC Methodology*, Technical Report ICS-99-56, University of California, Irvine, December 1999.

[3] R. Dömer, *The SpecC Internal Representation*, Technical Report ICS-99, University of California, Irvine, January 1999.

[4] Dan R. Olsen, Jr., *Developing User Interfaces*, San Francisco: Morgan Kauffmann Publishers, 1998.

[5] Trolltech AS, *QT: The Official Documentation*, New Riders Publishing, Devember 2000.

[6] Trolltech AS, *QT Online Reference Documentation v2.2.4*, http://doc.trolltech.com/index.html, January 2001.

[7] Matthias Kalle Dalheimer, *Programming with Qt*, Köln: O'Reilly, 1999.

[8] David M. Beazley, *SWIG Users Manual v1.1*, http://swig.sourceforge.net/doc.html, 1997.

[9] Guido van Rossum, Fred L. Drake, Jr., editor, *Python Reference Documentation v2.0*, http://www.python.org/doc/current, October 2000.

[10] Mark lutz and David Ascher, *Learning Python*, O'Reilly, March 1999.

[11] Bruce Eckel, *Thinking in C++*, New Jersey: Prentice Hall, 1995.

[12] Herbert Schildt, *C: The Complete Reference*, Berkeley: Osborne McGraw-Hill, 1987.

[13] Herbert Schildt, *Turbo C/C++*, Berkeley: Osborne McGraw-Hill, 1990.

A Communication with the Tools

A.1 Profiler

A.2 Architecture Refinement Tool

The communication between the user-interface and the architecture refinement tool is made through annotations in the SIR-datastructure. The datastructure itself is passed through an API between them. The following information has to be delivered in the specified format:

- Allocation Information
  The allocation information is a global annotation. It should look like this:
  _AR_PES = ”name1:proc1 name2:proc2 ... nameN:procN”
  where ”name1” is the name of the instance of processor one and ”proc1” is the the name of the processor-definition

- Partitioning Information
  The partitioning information is annotated in every behavior-instance. It is defined like this:
  _AR_MAPPEDTO = ”proc_name”
  ”proc_name” is the name of the processor the behavior-instance is mapped to. If there is no such annotation, the behavior-instance should be mapped to the processor it’s parent is mapped to.

- Scheduling Information
  In order to perform scheduling, the tool needs information about the order children of a behavior are to be executed on a processor.
  _AR_prname1 = ”inst1 inst2 ... instN”
  _AR_prname2 = ”inst1 inst2 ... instN”
  _AR_prname3 = ”inst1 inst2 ... instN”
  This means that for every processor there is an annotation with an ordered list of the children running on this processor. If there is a flattened hierarchy, inst1.inst1_1 and inst1.inst1_2 are used instead of inst1 (presuming inst1_1, inst1_2 are child-behaviors of inst1). The ”_AR_prnamex” - annotations do not exist in leaf-behaviors or flattened hierarchies since they only contain information about subbehaviors.

A.3 Communication Refinement-tool

The data-exchange with the communication refinement tool is made also through annotations in the SIR-datastructure. We try to make them as similar as possible to the architecture refinement annotations, only we use the prefix ”_CR_” for them.

- Allocation:
  List of the busses instantiated. ”Names” are the names of the instances, ”bus” are the names of the actual busses.
  _CR_BUSSES = ”name1:bus1 name2:bus2 ... nameN:busN”

- Channel mapping:
  Each channel-instance is annotated like this:
  ”_CR_BUS = busnamex”
  where ”busname” is the name of the appropriate bus-instance.

- Component-address:
  _CR_ADDRESS = (INT) address

- Addressing scheme:
  For each bus, there has to be specified an addressing-scheme

  _CR_DEST_START = (INT) first bit of destination address
  _CR_DEST_END = (INT) last bit of destination address
  _CR_SOURCE_START = (INT) first bit of source address
  _CR_SOURCE_END = (INT) last bit of source address
  _CR_MULTI_START = (INT) first bit of destination address
  _CR_MULTI_END = (INT) last bit of destination address

- Data transfer semantics:
  Each bus we annotate also with a data transfer semantics:

  _CR_SEND_ADDRESS = 1 or 0
  _CR_SEND_ID = 1 or 0
  _CR_SEND_TYPE = 1 or 0
  _CR_SEND_SIZE = 1 or 0

- Name of the protocol:
  _CR_PROTOCOL = protocolx

- Code of the protocol:
  Additionally in each bus the code for the specified protocol has to be included (can be retrieved from the protocol-database)

B Class Documentation

B.1 class allocation_imp - Enhances the Dialog *allocation*

Declared in module allocation_imp

B.1.1 Inheritance hierarchy:

allocation_imp.allocation_imp
allocation.allocation

B.1.2 Synopsis

```
class allocation_imp(allocation):
    def allocation_imp.allocation_imp.__init__(self, parent, label='Processor') #  Initi
    def allocation_imp.allocation_imp.add(self) #  Adds an item to the allocated -list.
    def allocation_imp.allocation_imp.alloc_changed(item) #  Is invoked when the selecti
    def allocation_imp.allocation_imp.avail_changed(item) #  Is invoked when the selecti
    def allocation_imp.allocation_imp.remove(self) #  Remove item from the allocated -li

    # Inherited from qt.QObject
    def qt.QObject.__init__(self, *args)

    # Inherited from qt.QWidget
    def qt.QWidget.__init__(self, *args)

    # Inherited from qt.QPaintDevice
    def qt.QPaintDevice.__del__(self)
    def qt.QPaintDevice.__init__(self, *args)

    # Inherited from qt.Qt
    def qt.Qt.__del__(self)
    def qt.Qt.__init__(self, *args)

    # Inherited from allocation.allocation
    def allocation.allocation.__init__(self, parent=None, name=None, modal=0, fl=0)
    def allocation.allocation.add(self)
    def allocation.allocation.alloc_changed(self, a0)
    def allocation.allocation.avail_changed(self, a0)
    def allocation.allocation.event(self, ev)
    def allocation.allocation.remove(self)
    def allocation.allocation.rename(self)

    # Inherited from qt.QDialog
    def qt.QDialog.__init__(self, *args)
```

B.1.3 Description

B.1.3.1 It offers a list of predefined processors with some data and let's the user allocate them for the design. Every instantiated processor (or bus) has to be named (the user will be prompted for a name).

self.available: list of available items

self.allocated: list of allocated items

self.add_button: add items to `allocated`

self.remove_button: remove items from `allocated`

self.label: defines the type of items which are listed (e.g. "processor")

B.1.4 allocation_imp.allocation_imp.add(self)

Adds an item to the `allocated` -list. If an item on the right side is marked, you will be asked for a name (dialog name_enter). The item will be added on the list on the right hand side.

B.1.5 allocation_imp.allocation_imp.alloc_changed(item)

Is invoked when the selection of `allocated` changes Will be used to enable/diable the remove-button. Not used yet

B.1.6 allocation_imp.allocation_imp.avail_changed(item)

Is invoked when the selection of `available` changes Will be used to enable/diable the add-button. Not used yet

B.1.7 allocation_imp.allocation_imp.remove(self)

Remove item from the `allocated` -list If no item is selected, an information-box will be shown, else the selected itme will be removed from the list.

B.2 class ApplicationWindow - The MDI Application-window

Declared in module RESpecCT

B.2.1 Inheritance hierarchy:

RESpecCT.ApplicationWindow
qt.QMainWindow

B.2.2 Synopsis

```
class ApplicationWindow(QMainWindow):
    def RESpecCT.ApplicationWindow.__init__(self) #  Initialize the main-window
    def RESpecCT.ApplicationWindow.about(self) #  Display an about-messagebox
    def RESpecCT.ApplicationWindow.aboutQt(self)
    def RESpecCT.ApplicationWindow.ar_refine(self) #  Run the partitioning-refinement-to
    def RESpecCT.ApplicationWindow.comm_refine(self) #  Call the communication refinemen
    def RESpecCT.ApplicationWindow.designUpdate(self) #  Update the display with the sir
    def RESpecCT.ApplicationWindow.edit_sc(self, fileName='', path='', line=0) #  Open a
    def RESpecCT.ApplicationWindow.findfile(self, dir, file) #  Find a SpecC-file in the
    def RESpecCT.ApplicationWindow.map_chnl(self) #  Map high-level-channels to allocate
    def RESpecCT.ApplicationWindow.map_pr(self, click) # Map a behavior to a processor
    def RESpecCT.ApplicationWindow.nop(self) #  Does nothing
    def RESpecCT.ApplicationWindow.openDesign(self, fileName=None) #  Opens a new design
    def RESpecCT.ApplicationWindow.profile(self) #  Run the profiler on the current desi
    def RESpecCT.ApplicationWindow.saveDesign(self, id=0) #  Shows a savedialog
    def RESpecCT.ApplicationWindow.select_busses(self) #  Do the allocation of busses
    def RESpecCT.ApplicationWindow.select_procs(self) #  Do the allocation of processors

    # Inherited from qt.QObject
    def qt.QObject.__init__(self, *args)

    # Inherited from qt.QMainWindow
    def qt.QMainWindow.__init__(self, *args)

    # Inherited from qt.QPaintDevice
    def qt.QPaintDevice.__del__(self)
    def qt.QPaintDevice.__init__(self, *args)

    # Inherited from qt.Qt
    def qt.Qt.__del__(self)
    def qt.Qt.__init__(self, *args)

    # Inherited from qt.QWidget
    def qt.QWidget.__init__(self, *args)
```

B.2.3 Description

This is the main-window. It contains a menu, treewindow and an mdi-workspace. And several toolbars. It manages all main objects like the behavior-tree and keeps references to them. Importand class-variables:

self.design: Reference to the actually displayed SIR_Design.

self.mbh: Reference to the mainbehavior of the design.

self.charts: List of open chartwidgets.

self.procs: List of allocated processors. Format: [('procname1','instname1'),(...),...]

self.busses: List of allocated busses. Format: [('busname1','instname1'),(...),...]

self.sc_edits: Dictionary of open SpecC code editor windows. Format: {}

self.sir: Reference to sc_tree-widget

self.ws: Reference to MDI-workspace

self.menuBar: Menubar

self.statusBar: Statusbar

B.2.4 RESpecCT.ApplicationWindow.__init__(self)

Initialize the main-window
All variables are initialized, the menu is filled, toolbars instantiated, the splitter, a tree and a QWorkspace are instantiated

B.2.5 RESpecCT.ApplicationWindow.ar_refine(self)

Run the partitioning-refinement-tool
To run this method, partitioning has to be performed and a top-level behaviors has to be set. After running it, the bahvior-tree will be updated with the resulting design

B.2.6 RESpecCT.ApplicationWindow.designUpdate(self)

Update the display with the sir-file in memory
Clears the behavior-tree and redisplays the contents of the design

B.2.7 RESpecCT.ApplicationWindow.edit_sc(self, fileName='', path='', line=0)

Open a SpecC-file in the code-editor
Takes a filename, an optional path and an optional linenumber if the file exists it is opened and displyed in a QMultiLineEdit The Caption of the titlebar of the editor will be the filename, the icon will be set to the specc-icon

B.2.8 RESpecCT.ApplicationWindow.findfile(self, dir, file)

Find a SpecC-file in the current directory
This method takes a QDir and a filename. The directory and it's subdiractories are successively searched for the specified filename. The comlete path of the first occurence is returned. if it is not found, an empty string is returned

B.2.9 RESpecCT.ApplicationWindow.map_chnl(self)

Map high-level-channels to allocated busses
Opens a *bus_map* -dialog in order to let the user map the top-level channels to the allocated busses.

B.2.10 RESpecCT.ApplicationWindow.map_pr(self, click)

Map a behavior to a processor

Gets called when the user clicks on a processor in the contextmenu of the behavior-tree. Determines which processor was selected, puts the name of that processor into the column and annotates the instance in the SIR_Design

B.2.11 RESpecCT.ApplicationWindow.nop(self)

Does nothing

Used for functions in the GUI which are not yet implemented (e.g. buttons or menu-items

B.2.12 RESpecCT.ApplicationWindow.openDesign(self, fileName=None)

Opens a new design

Uses QFileDialog; Loads SIR and sc-files, however if filename is specified, it is assumed to be a SIR-file

B.2.13 RESpecCT.ApplicationWindow.profile(self)

Run the profiler on the current design, display some general results

First the design is instrumented, then compiled and simulated, then profiled and eventually analysed with a certain weight-table. After all these steps have been successful, there are added some columns of general interest to the design (like traffic, operations and storage)

B.2.14 RESpecCT.ApplicationWindow.saveDesign(self, id=0)

Shows a savedialog

Lets the user save the design as SIR or SC

B.2.15 RESpecCT.ApplicationWindow.select_busses(self)

Do the allocation of busses

Opens an *allocation_imp* -dialog, adjusts all the captions and columnnames in the dialog, Insets some sample-data into the list of available busses. After closing the dialog, a list of the allocated busses is stored in *self.busses*.

B.2.16 RESpecCT.ApplicationWindow.select_procs(self)

Do the allocation of processors (achitecture exploration)

Opens the processor-allocation dialog (allocation_imp) and let's the user select processors. After the dialog is closed, a column "PE" is added to the tree (if not already existent and the allocated processors are inserted into the context-menu in order to give the user the possibility to perform partitioning.

B.3 class bus_map

Declared in module bus_map

B.3.1 Inheritance hierarchy:

bus_map.bus_map
allocation.allocation

B.3.2 Synopsis

```
class bus_map(allocation):
    def bus_map.bus_map.__init__(self, parent)
    def bus_map.bus_map.add(self)
    def bus_map.bus_map.alloc_changed(item)
    def bus_map.bus_map.avail_changed(item)
    def bus_map.bus_map.remove(self)

    # Inherited from qt.QObject
    def qt.QObject.__init__(self, *args)

    # Inherited from qt.QWidget
    def qt.QWidget.__init__(self, *args)

    # Inherited from qt.QPaintDevice
    def qt.QPaintDevice.__del__(self)
    def qt.QPaintDevice.__init__(self, *args)

    # Inherited from qt.Qt
    def qt.Qt.__del__(self)
    def qt.Qt.__init__(self, *args)

    # Inherited from allocation.allocation
    def allocation.allocation.__init__(self, parent=None, name=None, modal=0, fl=0)
    def allocation.allocation.add(self)
    def allocation.allocation.alloc_changed(self, a0)
    def allocation.allocation.avail_changed(self, a0)
    def allocation.allocation.event(self, ev)
    def allocation.allocation.remove(self)
    def allocation.allocation.rename(self)

    # Inherited from qt.QDialog
    def qt.QDialog.__init__(self, *args)
```

B.4 class SC_item - Itemclass for the SC_tree

Declared in module spec_tree

B.4.1 Inheritance hierarchy:

spec_tree.SC_item
qt.QListViewItem

B.4.2 Synopsis

```
class SC_item(QListViewItem):
    def spec_tree.SC_item.__init__(self, parent, inst, name) #  Initialize the Item
    def spec_tree.SC_item.arSchAnnotate(self)
    def spec_tree.SC_item.changeBeh(self, ask) #  Change the SIR_behavior of the item.
    def spec_tree.SC_item.fill_column(self, column, name) #  fill the column with the an
    def spec_tree.SC_item.getItemlist(self, Beh=None) #  Return a List with all Instance
    def spec_tree.SC_item.mappedTo(self) #  Returns the processor it is mapped to.

    # Inherited from qt.Qt
    def qt.Qt.__del__(self)
    def qt.Qt.__init__(self, *args)

    # Inherited from qt.QListViewItem
    def qt.QListViewItem.__init__(self, *args)
```

B.4.3 Description

Extends QlistViewItem with the functionality of displaying and handling with SIR_BhvrInstances. Class Variables:

self.Inst: Reference to the currently displayed instance.

self.Beh: SIR_Behavior of currently displayed instance

self.proc: Processor this instance is mapped to (only after partitioning)

B.4.4 spec_tree.SC_item.__init__(self, parent, inst, name)

Initialize the Item
Chooses the corresponding icon for the Item. Creates all child-items

parent: parent-item

inst: instance to be displayed

name: name of the instance

B.4.5 spec_tree.SC_item.changeBeh(self, ask)

Change the SIR_behavior of the item.
Changes the SIR_Behavior of the item. Performs the change not only in the SC_tree, but also in the SIR. Checks is there is multiple instantiations of the parent-behavior. Asks if the change should be performed for all instantiations or only for this.

B.4.6 spec_tree.SC_item.fill_column(self, column, name)

fill the column with the annotated value of note_name

column: is the number of the column the value should be added

name: name of the annotation

B.4.7 spec_tree.SC_item.getItemlist(self, Beh=None)

Return a List with all Instances of a Behavior
Uses SC_tree.beh_dic to retrieve the information, returns a list of SC_items

B.4.8 spec_tree.SC_item.mappedTo(self)

Returns the processor it is mapped to.
If the item is mapped to no processor, the mapping of the parent applies.

B.5 class SC_tree - Tree of Sir-behavior instances

Declared in module spec_tree

B.5.1 Inheritance hierarchy:

spec_tree.SC_tree
qt.QListView

B.5.2 Synopsis

```
class SC_tree(QListView):
    def spec_tree.SC_tree.__init__(self, parent) #  Initialize the SC_tree.
    def spec_tree.SC_tree.arSchAnnotate(self) #  Annotate the Design with the scheduling
    def spec_tree.SC_tree.clear(self) #  Clean up tree
    def spec_tree.SC_tree.col_add(self, note_name, name=None) #  Add a profiling-column
    def spec_tree.SC_tree.popup(self, item, point, col) #  Display popup menu
    def spec_tree.SC_tree.readSC(self, file) #  Display hierarchy of an SIR_Design
    def spec_tree.SC_tree.readSIR(self, file) #  Display hierarchy of an SIR_Design
    def spec_tree.SC_tree.updateSelected(self) #  Keep list of selected items correct

    # Inherited from qt.Qt
    def qt.Qt.__del__(self)
    def qt.Qt.__init__(self, *args)

    # Inherited from qt.QListView
    def qt.QListView.__init__(self, *args)

    # Inherited from qt.QFrame
    def qt.QFrame.__init__(self, *args)
```

```
    # Inherited from qt.QObject
    def qt.QObject.__init__(self, *args)

    # Inherited from qt.QPaintDevice
    def qt.QPaintDevice.__del__(self)
    def qt.QPaintDevice.__init__(self, *args)

    # Inherited from qt.QScrollView
    def qt.QScrollView.__init__(self, *args)

    # Inherited from qt.QWidget
    def qt.QWidget.__init__(self, *args)
```

B.5.3 Description

Extends QListview with the capabilities of reading and displaying the behavioral hierarchy of SIR_Design's. It assigns different icons to different types of behaviors and has a popup-menu with options to display and modify the behaviors. Class Variables:

self.beh_dic: Dictionary of all contained behaviors of form {'beh_name1':SC_item1,...}. Mainly in order to deal with a restriction of PyQt v 2.2 and earlier, that references to subclassed c-objects have to be kept.

self.column_dic: Dictionary containing all columns currently displaed in the tree. It has the form {'col_name1':(int)col_number, ...}. It is useful in order to check if a column exists and ant which position.

self.app: Reference to the Main-window

self.select_list: List of selected SC_items

B.5.4 spec_tree.SC_tree.__init__(self, parent)

Initialize the SC_tree.
Allows selection of multiple behaviors, disable sorting.

B.5.5 spec_tree.SC_tree.arSchAnnotate(self)

Annotate the Design with the scheduling-decisions made in the GUI
this is not in use yet, code existend was originally for something else, but can perhaps be re-used

B.5.6 spec_tree.SC_tree.clear(self)

Clean up tree
This method overrides/extends the clean-method of QListView. Removes all displayed items and columns. Also cleans up "self.beh_dic" and "self.column_dic".

B.5.7 spec_tree.SC_tree.col_add(self, note_name, name=None)

Add a profiling-column in the behavior-tree

note_name: Name of the Annotation in the SIR. the profiling-prefix is added automatically (e.g. `operations` -> PR_operations

name: Name of the column to add. By default it will be the name of the annotation.

B.5.8 spec_tree.SC_tree.popup(self, item, point, col)

Display popup menu

This is the slot-function of the right-mouseclick event of the tree. It displays a popupmenu at the current position and sets `"self.app.item"` to the currently selected item.

B.5.9 spec_tree.SC_tree.readSC(self, file)

Display hierarchy of an SIR_Design

Takes a filename (including path) of a .SC-file, reads it and displays it. Only the Top-level behavior is loaded, then the class SC_item takes care of the rest of the design. Since the 'ReadSC'-function of SIR does not support preprocessor commands right now, the specc-file should not include any preprocessor commands or be already preprocessed. TODO: Include preprocessor-functionality.

B.5.10 spec_tree.SC_tree.readSIR(self, file)

Display hierarchy of an SIR_Design

Takes a filename (including path) of a .SIR-file, reads it and displays it. Only the Top-level behavior is loaded, then the class SC_item takes care of the rest of the design.

B.5.11 spec_tree.SC_tree.updateSelected(self)

Keep list of selected items correct

This methos is a slot for selection_change. updates the list of selected items in order to provide it for the convenience of other methods.

B.6 class scalestruct - Small helperclass for scaling

Declared in module bar_chart

B.6.1 Synopsis

```
class scalestruct:
    def bar_chart.scalestruct.__init__(self, i=0, l=0)
```

B.6.2 Description

contains only two values: intervall of the scale and limitvalue which is the maximal value of the scale

B.7 class QxBarChart - Barchart with arbitrary number of columns and rows

Declared in module bar_chart

B.7.1 Inheritance hierarchy:

bar_chart.QxBarChart
qt.QWidget

B.7.2 Synopsis

```
class QxBarChart(QWidget):
    def bar_chart.QxBarChart.__init__(self, parent=None, Chartdata=0, Style=1, name='',
    def bar_chart.QxBarChart.close(self, bool) #  Close and delete the Chart
    def bar_chart.QxBarChart.doGeometry(self, P) #  Calculate the dimensions of the cont
    def bar_chart.QxBarChart.drawChartData(self, P) #  Draw the actual bars
    def bar_chart.QxBarChart.drawHorizontalLines(self, P) #  Draw horizontal lines to re
    def bar_chart.QxBarChart.drawLegends(self, P) #  Draws the legends for the chart.
    def bar_chart.QxBarChart.drawScale(self, P) #  Draws the x and y-scales and the grid
    def bar_chart.QxBarChart.drawTitles(self, P) # .
    def bar_chart.QxBarChart.drawXLegends(self, P) #  Draw the Legends for the X-axis
    def bar_chart.QxBarChart.paintEvent(self, PaintEvent) #  On every PaintEvent the dis
    def bar_chart.QxBarChart.setChartData(self, Chartdata) #  Set a new Chartdata

    # Inherited from qt.QObject
    def qt.QObject.__init__(self, *args)

    # Inherited from qt.Qt
    def qt.Qt.__del__(self)
    def qt.Qt.__init__(self, *args)

    # Inherited from qt.QPaintDevice
    def qt.QPaintDevice.__del__(self)
    def qt.QPaintDevice.__init__(self, *args)

    # Inherited from qt.QWidget
    def qt.QWidget.__init__(self, *args)
```

B.7.3 Description

Best display not more than 5 rows an 15 columns are recommended (depending on the size on the screen you choose).

B.7.4 bar_chart.QxBarChart.__init__(self, parent=None, Chartdata=0, Style=1, name='', f=0)

Initialises the Chart

parent parent-widget on which it will be centered

Chartdata QxChartData conaining the data, labels and Fonts

Style show legends on x-axis or not

B.7.5 bar_chart.QxBarChart.close(self, bool)

Close and delete the Chart
First call the close-method of QWidget, then deletes the widget. It also removed the chart form the list of charts held in the Application

B.7.6 bar_chart.QxBarChart.doGeometry(self, P)

Calculate the dimensions of the content

- self.titleRect
- self.subTitleRect
- self.footerRect
- self.scaleRect
- self.chartDataRect
- self.xlegendsRect
- self.legendsRect

B.7.7 bar_chart.QxBarChart.drawChartData(self, P)

Draw the actual bars

- P is the painter where the data is painted into.
- To draw the bars in the chart, we first detect where the zeroline will be.
- The scaling of the bars is obtained from self.qxScale.
- The colors for the series are taken consecutively from the qxColorlist.
- Between two series there will be a small space

B.7.8 bar_chart.QxBarChart.drawHorizontalLines(self, P)

Draw horizontal lines to reflect the scale accross the entire barchart

- The number of lines is equal to (number of scale items
- 1).
- We iterate bottom up and draw a Draw line across the drawing area for the barchart.
- The QxScale method `valueScaleRatio(it)` determines the ratio for the particular item in y space.
- We need to iterate through the scalevalues,and set the y-coordinate to self.qxScale.valueScaleRatio(it) * height of the datarect.

B.7.9 bar_chart.QxBarChart.drawLegends(self, P)

Draws the legends for the chart.
`legendsRect.height` is split into ten if the number of series are less than ten, else it is split into the numer of columns. The colors are taken consecutively from `QxCommonColor`

B.7.10 bar_chart.QxBarChart.drawScale(self, P)

Draws the x and y-scales and the grid
If on the Y-axis the value is > 1000 or the step of the scaple is $< 1/100$, the value is displayed in scientific mode (e.g. `1.3 e-5`), if $|\text{value}|$ is < 1000, the display is absolute

B.7.11 bar_chart.QxBarChart.drawTitles(self, P)

.
Draw the Title, Subtitle and Footer

B.7.12 bar_chart.QxBarChart.drawXLegends(self, P)

Draw the Legends for the X-axis

- Loop from left to right;
- Spacing is width / len(xlegends).
- Split xlegendsRect into len(xlegends) rects

B.7.13 bar_chart.QxBarChart.paintEvent(self, PaintEvent)

On every PaintEvent the display is refreshed
Updates the geometry of the widget and then repaints all of it's contents (data, legends and captions)

B.7.14 bar_chart.QxBarChart.setChartData(self, Chartdata)

Set a new `Chartdata`
If it is None, a new `QxChartData` -instance is created. Initializes `qxscale` and draws the chart for the first time

B.8 class QxChartData - Contains all the data for the Chart

Declared in module bar_chart

B.8.1 Synopsis

```
class QxChartData:
    def bar_chart.QxChartData.__init__(self, data=[[400, 80, 150], [111, 270, 543]], row
    def bar_chart.QxChartData.cols(self) #  Return number of columns
    def bar_chart.QxChartData.getColumnName(self, pos)
    def bar_chart.QxChartData.getFooter(self)
    def bar_chart.QxChartData.getFooterFont(self)
    def bar_chart.QxChartData.getLegendsFont(self)
```

```
    def bar_chart.QxChartData.getRowName(self, pos)
    def bar_chart.QxChartData.getSubTitle(self)
    def bar_chart.QxChartData.getSubTitleFont(self)
    def bar_chart.QxChartData.getTitle(self)
    def bar_chart.QxChartData.getTitleFont(self)
    def bar_chart.QxChartData.getValue(self, c, r)
    def bar_chart.QxChartData.getXLegendsFont(self)
    def bar_chart.QxChartData.highestValue(self) #  Return the highest value of the char
    def bar_chart.QxChartData.lowestValue(self) #  Return the lowest value of the chart
    def bar_chart.QxChartData.rows(self) #  Return number of rows
    def bar_chart.QxChartData.setFooter(self, footer, font)
    def bar_chart.QxChartData.setLegendsFont(font)
    def bar_chart.QxChartData.setSubTitle(self, title, font)
    def bar_chart.QxChartData.setTitle(self, title, font=0)
    def bar_chart.QxChartData.setXLegendsFont(font)
```

B.8.2 Description

self.data: Is of the form [[r1_1,r1_2,r1_3],[r2_1,r2_2,r2_3]]

self.rowlabels: [row_l1,row_l2]

self.col_labels: [col_l1,col_l2]

titles: Strings for title, subtitle and footer

fonts: Every label has a font

B.8.3 bar_chart.QxChartData.__init__(self, data=[[400, 80, 150], [111, 270, 543]], row_labels=['breakfast', 'lunch'], col_labels=['spam', 'egg', 'ham'], title='')

Constructor takes data, row_labels, col_labels and title

all other data has to be set either directly: 'cd.subTitle='spanish" or through the set-methods: `cd.setSubTitle('inquisition')`

B.9 class QxPie - Class representing the pie of the widget.

Declared in module piewidget

B.9.1 Synopsis

```
class QxPie:
    def piewidget.QxPie.__init__(self) #  Constructor
    def piewidget.QxPie.append(self, slice) #  Append slice
    def piewidget.QxPie.arcLength(self, index) #  Lenght of arc
    def piewidget.QxPie.arcStart(self, index) #  Start of the arc
    def piewidget.QxPie.at(self, pos) #  .
    def piewidget.QxPie.count(self) #  Count slices
    def piewidget.QxPie.insert(self, pos, slice) #  Insert slice
```

```
def piewidget.QxPie.sliceRatio(self, index) #  Relative value of slice
def piewidget.QxPie.sliceRatioAsPercentageString(self, index) #  Ratio as percentage
def piewidget.QxPie.sliceRatioAsString(self, index, precision=2)
```

B.9.2 Description

It contains a list of slices (self.list) containing the actual data and a number of accessor-methods which calculate geometrical results.

B.9.3 piewidget.QxPie.__init__(self)

Constructor
Initialize the list.

B.9.4 piewidget.QxPie.append(self, slice)

Append slice
Append a slice at the end of the list.

B.9.5 piewidget.QxPie.arcLength(self, index)

Lenght of arc
Returns the actual length of the arc of the slice at position `index` in the list

B.9.6 piewidget.QxPie.arcStart(self, index)

Start of the arc
Returns the position where the arc of the slice at `index` starts.

B.9.7 piewidget.QxPie.at(self, pos)

.
Return the slice at the position `pos`.

B.9.8 piewidget.QxPie.count(self)

Count slices
Return the number of slices.

B.9.9 piewidget.QxPie.insert(self, pos, slice)

Insert slice
Insert a slice at position `pos`.

B.9.10 piewidget.QxPie.sliceRatio(self, index)

Relative value of slice
Returns the ration of the sum of all slices and the value of the slice at `index` Example: slice1=5 slice2=10 slice3=15; ration of slice2 is 10 / (5+10+15)= 1/3; slice 2 occupies one third of the circle (=120 degree).

B.9.11 piewidget.QxPie.sliceRatioAsPercentageString(self, index)

Ratio as percentage
Same as sliceRatio, only that it does return a string representing the percentage, not the actual ratio.

B.10 class QxPieWidget - Pie-Widget class.

Declared in module piewidget

B.10.1 Inheritance hierarchy:

piewidget.QxPieWidget
qt.QWidget

B.10.2 Synopsis

```
class QxPieWidget(QWidget):
    def piewidget.QxPieWidget.__init__(self, parent=0, name=0, f=0, pie=0, align=1, show
    def piewidget.QxPieWidget.addSlice(self, slice, pos) #  Add a slice to the pie
    def piewidget.QxPieWidget.close(self, bool) #  Close the Widget
    def piewidget.QxPieWidget.doGeometry(self) # Calculate the geometry of the pie
    def piewidget.QxPieWidget.drawLegends(self, P) #  .
    def piewidget.QxPieWidget.drawSlices(self, P) #  .
    def piewidget.QxPieWidget.drawText(self, P) #  .
    def piewidget.QxPieWidget.drawTitle(self, P) #  .
    def piewidget.QxPieWidget.explodeFlag(self, explode) #  Set the distance between sli
    def piewidget.QxPieWidget.explodePoint(self, c) #  "explode" the pie.
    def piewidget.QxPieWidget.legendsAlignFlag(self, align) #  Set the alignment of the
    def piewidget.QxPieWidget.paintEvent(self, paintev) #  Repaint the pie
    def piewidget.QxPieWidget.resizeEvent(self, resizeEV) #  .
    def piewidget.QxPieWidget.setPie(self, pie) #  Insert a new pie
    def piewidget.QxPieWidget.set_data(self, data, title='', subtitle='', footer='', leg
    def piewidget.QxPieWidget.showFlag(self, show) #  Set data shown in slice
    def piewidget.QxPieWidget.signalStyleChanged(self, str)

    # Inherited from qt.QObject
    def qt.QObject.__init__(self, *args)

    # Inherited from qt.Qt
    def qt.Qt.__del__(self)
    def qt.Qt.__init__(self, *args)

    # Inherited from qt.QPaintDevice
    def qt.QPaintDevice.__del__(self)
    def qt.QPaintDevice.__init__(self, *args)

    # Inherited from qt.QWidget
    def qt.QWidget.__init__(self, *args)
```

B.10.3 Description

Class for displaying pie-charts.

B.10.4 piewidget.QxPieWidget.__init__(self, parent=0, name=0, f=0, pie=0, align=1, show=64, explode=128)

Constructor for pie-widgets
...

B.10.5 piewidget.QxPieWidget.addSlice(self, slice, pos)

Add a slice to the pie

slice: a QXslice-object

pos: the position where the slice is to be inserted (integer)

B.10.6 piewidget.QxPieWidget.close(self, bool)

Close the Widget
first call widgets-close-method, then delete widget

B.10.7 piewidget.QxPieWidget.doGeometry(self)

Calculate the geometry of the pie
Determines hom much space for every element in the widget is there and positions them.

B.10.8 piewidget.QxPieWidget.drawLegends(self, P)

.
Draw the legends of the pie.

B.10.9 piewidget.QxPieWidget.drawSlices(self, P)

.
Draw the slices of the pie (actual data). the colors are takes succesively from qxColorlist.

B.10.10 piewidget.QxPieWidget.drawText(self, P)

.
Draw the text of the pie. Evaluates the show-flag.

B.10.11 piewidget.QxPieWidget.drawTitle(self, P)

.
Draws the titles of the pie (title, subtitle and footer). The specific font sets apply.

B.10.12 piewidget.QxPieWidget.explodeFlag(self, explode)

Set the distance between slices
The explode-flag means, that the slices ale slightly pulled out, which might result in a better overview. Possible values are:

B.10.13 piewidget.QxPieWidget.explodePoint(self, c)

"explode" the pie.
Evaluates the explode-flag

B.10.14 piewidget.QxPieWidget.legendsAlignFlag(self, align)

Set the alignment of the legends

align: flag with possible values: QxLegendsToLeft, QxLegendsToRight or QxNoLegends

B.10.15 piewidget.QxPieWidget.paintEvent(self, paintev)

Repaint the pie
redraws the title, slices, legends and the text.

B.10.16 piewidget.QxPieWidget.resizeEvent(self, resizeEV)

.
Resize the pie and update its geometry.

B.10.17 piewidget.QxPieWidget.setPie(self, pie)

Insert a new pie
The old pie is saved as `self.oldPie`

B.10.18 piewidget.QxPieWidget.set_data(self, data, title='', subtitle='', footer='', legendstitle='')

initializes the pie with data and text and displays it
data is a list of value-pairs like [("slicename1", value1),("slice2", v2) ...] labels is a list of three strings: ['title','subtitle','footer'] font is the font for all the three labels

B.10.19 piewidget.QxPieWidget.showFlag(self, show)

Set data shown in slice
Inside a slice can be displayed:

QxShowSliceRatio: QxShowValues: – QxShowLabels: – QxShowPercentage: -

QxShowValues: QxShowLabels: – QxShowPercentage: -

QxShowLabels: QxShowPercentage: –

B.11 class QxScale - Create a scale between two given double numbers

Declared in module bar_chart

B.11.1 Synopsis

```
class QxScale:
    def bar_chart.QxScale.__init__(self, s=0, high=0) #  either none, one or two paramet
    def bar_chart.QxScale.count(self) #  Return the number of scalevalues
    def bar_chart.QxScale.createScale(self) #  Create a new scale
    def bar_chart.QxScale.getScale(self) #  Return self.scalevalues
    def bar_chart.QxScale.num_intervall(self, Highest) #  Determine the number of interv
    def bar_chart.QxScale.valueScaleRatio(self, it) #  Basically just determines the hei
    def bar_chart.QxScale.zeroLineRatio(self) #  Determine the position of the zeroline
```

B.11.2 Description

This class tries to make a scale for a set of values. The intervall and the maximal value of the scale are chosen such, that there is only 4-6 scalevalues with very few digits (e.g. 0.1, 0.2, 0.3, 0.4). For creating a scale we distinguish tree cases:

CASE 1: -x1.....0.....+x2

Creates an appropriate scale from -x1 to +x2. If $|-x1| > x2$ the negative side determines the scaling. If x2 $>= |-x1|$ the positive side determines the scaling.

CASE 2: 0.....+x2

Scaling done from 0 to x2

CASE 3: -x1.....0

Scaling done from -x1 to 0.

B.11.3 bar_chart.QxScale.__init__(self, s=0, high=0)

either none, one or two parameters

none: everything =0.0

one: self.scalevalues is initialized

two: low, high

B.11.4 bar_chart.QxScale.createScale(self)

Create a new scale

Clear first the scalevalues list, just in case we have run this before, then fill it up. Here the values of `self.highest` and `self.lowest` are evaluated

B.11.5 bar_chart.QxScale.num_intervall(self, Highest)

Determine the number of intervalls needed

Attempts to choose the number of intervalls such, that the intervalls are as even as possible returns a scalestruct (pair of intervall and limitvalue)

B.11.6 bar_chart.QxScale.valueScaleRatio(self, it)

Basically just determines the height of the bar
`it` is the index of the scalevalue to be examined. The method returns the (relative) height of the resulting bar.

B.11.7 bar_chart.QxScale.zeroLineRatio(self)

Determine the position of the zeroline

- If beginning is equal to zero, the ratio is 1
- If end is equal to zero, the ratio is 0
- If zero is found somewhere in between: 1-(beg/(beg+end))

B.12 class QxSlice - Slice of a pie

Declared in module piewidget

B.12.1 Synopsis

```
class QxSlice:
    def piewidget.QxSlice.__init__(self, v=0, label=0) #  Constructor for a Slice.
    def piewidget.QxSlice.setLabel(self, label) #  .
    def piewidget.QxSlice.setValue(self, v) #  .
    def piewidget.QxSlice.value(self) #  .
    def piewidget.QxSlice.valueString(self, precision=2) #  return a string representign
```

B.12.2 Description

Represents a slice of a pie. It contains a value and a label.

B.12.3 piewidget.QxSlice.__init__(self, v=0, label=0)

Constructor for a Slice.
Assigns an initial value and a label

B.12.4 piewidget.QxSlice.setLabel(self, label)

.
Assign a new label to the slice

B.12.5 piewidget.QxSlice.setValue(self, v)

.
Assign a new value to the slice

B.12.6 piewidget.QxSlice.value(self)

.

Access the variable `Value`

B.12.7 piewidget.QxSlice.valueString(self, precision=2)

return a string representign the value

`precision` optionally specifies the precision with which floating-point numbers are converted. Default is 2 digits.

C Code-examples

C.1 Header-file of SIR_Behavior

```
/*************************************************************************/
/* IntRep/Behavior.h: SpecC Internal Representation, Level 2, Behaviors */
/*************************************************************************/
/* Author: Rainer Doemer                          first version: 07/06/98 */
/*************************************************************************/
/* last update: 04/12/99 */

#ifndef INTREP_BEHAVIOR_H
#define INTREP_BEHAVIOR_H

#include "Global.h"
#include "IntRep/Class.h"

/*** enumeration types ***************************************************/

enum SIR_BehaviorClass /* supported behavior classifications */
{
SIR_BHVR_EXTERN,    /* external behavior (black box) */
SIR_BHVR_LEAF,      /* leaf behavior */
SIR_BHVR_SEQ,       /* "clean" sequential behavior */
SIR_BHVR_PAR,       /* "clean" concurrent behavior (par {}) */
SIR_BHVR_PIPE,      /* "clean" pipelined behavior (pipe {}) */
SIR_BHVR_FSM,       /* "clean" FSM-style behavior (fsm {}) */
SIR_BHVR_TRY,       /* "clean" exception-handling behavior (try {}) */
SIR_BHVR_OTHER      /* "dirty" compound behavior */
};

/*** type definitions ****************************************************/

typedef enum SIR_BehaviorClass      SIR_BHVR_CLASS;
typedef class SIR_Behavior          sir_behavior;
typedef SIR_List<sir_behavior>      sir_behavior_list;
typedef class SIR_Behaviors         sir_behaviors;
typedef ERROR                       (*sir_bhvr_fct)(sir_behavior*, void*);
typedef class SIR_Design            sir_design;     /* cyclic link */

/*** class declarations **************************************************/

        /********************/
        /*** SIR_Behavior ***/
        /********************/

class SIR_Behavior :                /* behavior class in hierarchy tree */
        public SIR_ListElem<SIR_Behavior>,    /* is a list element */
```

```
        public SIR_Class                        /* and a class */
{
public:
SIR_BHVR_CLASS BehaviorClass; /* behavior classification (see above) */
 sir_function      *MainMethod;    /* link to main method (or NULL) */
 sir_statement     *FirstBhvrCall; /* link to first subbehavior call (or NULL) */
 sir_statement     *BehaviorCalls; /* link to cmpnd.stmnt. with behavior calls */
                                   /* (if SEQ or PAR or PIPE, else NULL) */
 sir_transitions   *Transitions;   /* link to transitions (if FSM, else NULL) */
 sir_exceptions    *Exceptions;    /* link to exceptions (if TRY, else NULL) */

//++++++++++++++++++++++++++++ API Layer 1 ++++++++++++++++++++++++++++//

SIR_Behavior(                     /* constructor #1 (initial) */
        sir_symbol        *Symbol);

~SIR_Behavior(void);              /* destructor */
void FinishConstruction(void);    /* perform construction phase 2 */
void UpdateInfos(void);           /* update classification and links */
static sir_behavior *GetBehavior(       /* obtain behavior pointer (level 2) */
        sir_symbol        *Symbol);     /* from a behavior symbol */

//++++++++++++++++++++++++++++ API Layer 2 ++++++++++++++++++++++++++++//

static sir_behavior *Create(      /* create a new behavior */
        const char        *Name,  /* (returns NULL if SIR_Error) */
         sir_design       *Design,
        BOOL               Internal = FALSE);     /* default: without body */

sir_behavior *Copy(               /* create an exact copy with new name */
        const char        *Name,  /* (returns NULL if SIR_Error) */
        BOOL               Strip = TRUE);

ERROR Rename(                     /* rename this behavior */
        const char        *Name);

ERROR Delete(void);               /* delete this behavior */
sir_behaviors *GetList(void);     /* obtain a pointer to the behavior list */
SIR_BHVR_CLASS GetClass(void); /* obtain this behaviors classification */
BOOL FindInstance(                /* find an instance of this behavior */
         sir_bhvrinst     **BhvrInst = NULL,    /* return instance found */
         sir_bhvrinst     *LastInstance = NULL); /* continue search here */

ERROR CreateBody(void);           /* create a minimal behavior body */
ERROR DeleteBody(void);           /* delete the behavior body */
                                  /* (so that it becomes an external behavior) */

ERROR MakeMainMethod(             /* generates a main method (template) */
```

```
        SIR_BHVR_CLASS BehaviorClass,        /* intended behavior class */
94          sir_bhvrinst     *FirstBhvr = NULL);    /* first subbehavior */
};
96
        /*********************/
98      /*** SIR_Behaviors ***/
        /*********************/
100
class SIR_Behaviors :               /* behavior classes list */
102     public SIR_List<SIR_Behavior> /* is simply a list of behaviors */
{                                   /* with additional methods */
104 public:

106 //+++++++++++++++++++++++++++++ API Layer 1 +++++++++++++++++++++++++++++//

108 SIR_Behaviors(                   /* constructor #1 */
        sir_behavior     * FirstEntry = NULL);
110
~SIR_Behaviors(void);               /* destructor */
112
static sir_behaviors * BuildList (  /* build the list of behaviors */
114     sir_symbols      *GlobalSymbols);       /* ( phase 1) */

116 void FinishConstruction (void);            /* perform construction phase 2 */

118 sir_behavior * Insert (             /* insert a prepared element */
        sir_behavior     *Behavior);
120
//+++++++++++++++++++++++++++++ API Layer 2 +++++++++++++++++++++++++++++//
122
sir_behavior *Find(                 /* find a behavior with this name */
124     const char       *Name);        /* ( returns NULL if not found ) */
};
126
#endif /* INTREP_BEHAVIOR_H */
128 /* EOF IntRep/Behavior.h */
```

C.2 Interface-file of SIR_Behavior

```
/* SWIG - Header-file for Python Wrapper
 File :             Behavior.i
 Date generated : 12/5/2000    18:17h
4 Author:          David Berner */

6 %module Behavior

8 %{
#include "IntRep/Behavior.h"
```

```
10  #include "Global.h"
    #include "IntRep/Class.h"
12  %}

14  %import Class.i

16  enum SIR_BehaviorClass /* supported behavior classifications */
    {
18  SIR_BHVR_EXTERN,    /* external behavior (black box) */
    SIR_BHVR_LEAF,      /* leaf behavior */
20  SIR_BHVR_SEQ,       /* "clean" sequential behavior */
    SIR_BHVR_PAR,       /* "clean" concurrent behavior (par {}) */
22  SIR_BHVR_PIPE,      /* "clean" pipelined behavior (pipe {}) */
    SIR_BHVR_FSM,       /* "clean" FSM-style behavior (fsm {}) */
24  SIR_BHVR_TRY,       /* "clean" exception-handling behavior (try {}) */
    SIR_BHVR_OTHER      /* "dirty" compound behavior */
26  };

28  /*** type definitions ***********************************************/

30  typedef enum SIR_BehaviorClass      SIR_BHVR_CLASS;
    typedef class SIR_Behavior          sir_behavior;
32  typedef SIR_List<sir_behavior>      sir_behavior_list;
    typedef class SIR_Behaviors         sir_behaviors;
34  typedef ERROR                       (*sir_bhvr_fct)(sir_behavior*, void*);
    typedef class SIR_Design            sir_design;     /* cyclic link */
36
    /*** class declarations *********************************************/
38
        /********************/
40      /*** SIR_Behavior ***/
        /********************/
42
    class SIR_Behavior :            /* behavior class in hierarchy tree */
44      public SIR_Class                        /* and a class */
    {
46  public:
    unsigned long UnitID;
48
    // automatic template-path insert begin (list-element):
50
    SIR_Behavior *Succ(void);       /* Successor */
52  SIR_Behavior *Pred(void);               /* Predecessor */
    SIR_Behaviors *Head(void);      /* List head */
54  void Remove(void);              /* remove myself */

56  // automatic template-path insert end (list-element)
```

```
58  SIR_BHVR_CLASS BehaviorClass; /* behavior classification (see above) */
    sir_function     *MainMethod;    /* link to main method (or NULL) */
60  sir_statement    *FirstBhvrCall; /* link to first subbehavior call (or NULL) */
    sir_statement    *BehaviorCalls; /* link to cmpnd.stmnt. with behavior calls */
62                                   /* (if SEQ or PAR or PIPE, else NULL) */
    sir_transitions  *Transitions;   /* link to transitions (if FSM, else NULL) */
64  sir_exceptions   *Exceptions;    /* link to exceptions (if TRY, else NULL) */

66  //++++++++++++++++++++++++++++ API Layer 2 ++++++++++++++++++++++++++++//

68  static SIR_Behavior *Create(     /* create a new behavior */
            const char      *Name,   /* (returns NULL if SIR_Error) */
70          sir_design      *Design,
            BOOL            Internal = FALSE);     /* default: without body */
72
    SIR_Behavior *Copy(              /* create an exact copy with new name */
74          const char      *Name,   /* (returns NULL if SIR_Error) */
            BOOL            Strip = TRUE);
76
    ERROR Rename(                    /* rename this behavior */
78          const char      *Name);

80  ERROR Delete(void);              /* delete this behavior */

82  SIR_Behaviors *GetList(void);    /* obtain a pointer to the behavior list */
    SIR_BHVR_CLASS GetClass(void); /* obtain this behaviors classification */
84
    BOOL FindInstance(               /* find an instance of this behavior */
86          SIR_BhvrInst    **BhvrInst = NULL,     /* return instance found */
            SIR_BhvrInst    *LastInstance = NULL); /* continue search here */
88
    ERROR CreateBody(void);          /* create a minimal behavior body */
90                                   /* (so that it becomes an internal behavior) */

92  ERROR DeleteBody(void);          /* delete the behavior body */
                                     /* (so that it becomes an external behavior) */
94
    ERROR MakeMainMethod(            /* generates a main method (template) */
96          SIR_BHVR_CLASS BehaviorClass,        /* intended behavior class */
            sir_bhvrinst    *FirstBhvr = NULL);  /* first subbehavior */
98  };

100         /*********************/
            /*** SIR_Behaviors ***/
102         /*********************/

104 class SIR_Behaviors              /* behavior classes list */
    {                                        /* with additional methods */
```

```
106 public:

108  // automatic template-path insert begin: ( list )

110 bool Empty(void);                    /* test for empty list ? */
    unsigned int NumElements(void); /* number of list elements */
112 SIR_Behavior * First (void);                /* first element (NULL if empty) */
    SIR_Behavior *Last(void);                  /* last element (NULL if empty) */
114 SIR_Behavior *Previous(void);     /* previous element (NULL if none) */
    SIR_Behavior *Curr(void);                  /* current element (NULL if none) */
116 SIR_Behavior *Next(void);                  /* next element (NULL if none) */
    SIR_Behavior *Prepend(SIR_Behavior        *Elem);
118 SIR_Behavior *Append(SIR_Behavior *Elem);
    SIR_Behavior * InsertBefore (SIR_Behavior *Elem,SIR_Behavior *Succ);
120 SIR_Behavior * InsertAfter (SIR_Behavior *Elem,SIR_Behavior *Pred);
    SIR_Behavior *Remove(SIR_Behavior *Elem);
122 SIR_Behaviors *Concat(SIR_Behaviors *Appendix);
    SIR_Behaviors * Precat (SIR_Behaviors *Prependix);
124
     // automatic template-path insert end ( list )
126
    //++++++++++++++++++++++++++++ API Layer 2 ++++++++++++++++++++++++++++//
128
     sir_behavior *Find(                    /* find a behavior with this name */
130          const char       *Name);          /* ( returns NULL if not found) */
    };
```

C.3 SWIG interface-file generator: template.py

```
    # template.py script for automatical generation of headerfiles for SWIG
2   # David Berner 11/30/2000


4
    import string , sys , time
6
    def find_template ():
8     tm = time. localtime (time.time ())
      tm_str = str (tm[1])+'/'+str (tm[2])+'/'+str (tm[0])+' '+str (tm[3])+':'+str (tm[4])
10    print tm_str
       outfile . write ('/* SWIG -- Header-file for Python Wrapper \n File:    '+sys.argv[2]+'.i\n Date generated: '+tm_str+'h
12    ln= infile . readline ()
       outfile . write (ln)
14    print sys.argv
      while not(ln == ''):
16        slist = ln. split ()
#       if len( slist ) > 1 and slist [0] == ' class ' and slist [2] == ':':
18  #         sir_type = slist [1]
    #         cl_type = get_cl ( slist )
```

```
#         patch( cl_type ,  sir_type , ln )
      ln= infile . readline ()
       outfile . write ( ln )
    a=0

def  patch( cl_type ,  sir_type , ln ):
    """ Remove the inheritance-statement and insert adapted method-prototypes"""
    print  '\npatching ', cl_type , ' in class ', sir_type , ''
    while not ( ln . split ()[0] ==  'public :' ):
       ln =  infile . readline ()
       outfile . write ( ln )
    if  cl_type  ==  ' list -element':
       outfile . write ('\n// automatic template-path insert begin ( list -element):\n\n')
       outfile . writelines ( sir_type ,  '*Succ(void );        /* Successor */\n')
       outfile . writelines ( sir_type ,  '*Pred(void );          /* Predecessor */\n')
       outfile . writelines ( sir_type , 's  *Head(void );       /* List head */\n')
       outfile . write ('void Remove(void);            /* remove myself */\n')
       outfile . write ('\n// automatic template-path insert end ( list -element)\n\n')

    elif  cl_type  ==  ' list ':
       outfile . write ('\n// automatic template-path insert begin : ( list )\n\n')
       outfile . write ('bool Empty(void);\t\t/* test for empty list ? */\n')
       outfile . write ('unsigned int NumElements(void);\t/* number of list elements */\n')
       outfile . write ( sir_type [:-1]+' * First (void );\t\t/* first element (NULL if empty) */\n')
       outfile . write ( sir_type [:-1]+' *Last(void );\t\t/* last element (NULL if empty) */\n')
       outfile . write ( sir_type [:-1]+' *Previous(void );\t/* previous element (NULL if none) */\n')
       outfile . writelines ( sir_type [:-1], ' *Curr(void );\t\t/* current element (NULL if none) */\n')
       outfile . writelines ( sir_type [:-1], ' *Next(void );\t\t/* next element (NULL if none) */\n')
       outfile . writelines ( sir_type [:-1], ' *Prepend(', sir_type [:-1], '\t *Elem);\n')
       outfile . writelines ( sir_type [:-1], ' *Append(', sir_type [:-1], '\t *Elem);\n')
       outfile . writelines ( sir_type [:-1], ' * InsertBefore (', sir_type [:-1], '\t *Elem,', sir_type [:-1], ' *Succ);\n')
       outfile . writelines ( sir_type [:-1], ' * InsertAfter (', sir_type [:-1], '\t *Elem,', sir_type [:-1], ' *Pred);\n')
#       outfile . writelines ( sir_type  [:-1],' * Remove(', sir_type  [:-1],' * Elem);\n')
#   the remove-method exists two times , one is removed :)
       outfile . writelines ( sir_type [:-1], ' *Remove(', sir_type [:-1], ' *Elem);\n')
       outfile . writelines ( sir_type , ' *Concat(', sir_type , ' *Appendix);\n')
       outfile . writelines ( sir_type , ' *Precat (', sir_type , ' *Prependix);\n')
       outfile . write ('\n// automatic template-path insert end ( list )\n\n')

def  get_cl ( slist ):
    """ Determine if the class is inherited fom a template-class and which """
     cl_tp =0
    while not ( slist  [0] ==  '{'):
       ln= infile . readline ()
        slist  =  ln . split ()
       if  ( not( string . find ( ln , 'SIR_ListElem<') == -1)):
          cl_tp =' list -element'
       elif  not ( string . find ( ln , 'SIR_List<') == -1):
```

```
68            cl_tp =' list '
          else :
70            outfile . write (ln)
      return  cl_tp
72

74  if  len(sys.argv) > 1:
      infile  = open(sys.argv[1], 'r')
76    outfile  = open(sys.argv[2]+'.i','w')
      a = find_template ()
78    infile . close ()
      outfile . close ()
80
    else :
82    print  "\nusage: python template.py header-file module-name"
```