CRefine: Support for the *Circus* Refinement Calculus

M. V. M. Oliveira
DIMAP/UFRN
Natal, Brazil
marcel@dimap.ufrn.br

A. C. Gurgel
DIMAP/UFRN
Natal, Brazil
agurgel2000@yahoo.com.br

C. G. Castro
DIMAP/UFRN
Natal, Brazil
crisgc1@gmail.com

Abstract

Circus specifications combine both data and behavioural aspects of concurrent systems using a combination of CSP, Z, and Dijkstra's command language. Its associated refinement theory and calculus distinguishes itself from other such combinations. Recently, tools for Circus like a parser, a type-checker, a model-checker, and a translator to Java have been developed. Nevertheless, tool support for the Circus refinement calculus has only been prototyped. In this paper, we present CRefine, a tool that can be used throughout the refinement of concurrent systems in a calculational approach. Its functionalities are described using a simple case study. Furthermore, we also describe CRefine's architecture and its integration to the CZT framework.

1. Introduction

Languages like Z [25], VDM-SL [21], and B [1], use a model-based approach to specification, based on mathematical objects from set theory. Although possible, modelling behavioural aspects such as choice, sequence, parallelism, and others, using these languages, is difficult and needs to be done in an implicit fashion. On the other hand, process algebras like CSP [10] and CCS [13] provide constructs that can be used to describe the behaviour of the system. However, they do not support a concise and elegant way to describe complex data aspects.

Many attempts to join these two kinds of formalism have been made. Combinations of Z with CCS [23], Z with CSP [22], and Object-Z with CSP [4] are some examples. The lack of support for refinement of state-rich reactive systems in a calculational style as that presented in [14, 7] has motivated the creation of *Circus* [18]. The semantics of *Circus* has already been mechanised [19]; it is based on the *Unifying Theories of Programming* [11].

In *Circus*, systems are characterised as processes, which group constructs that describe data and control behaviour; Z is used to define most of the data aspects, and CSP is used to define behaviour. Besides, the language provides support for formal stepwise development of concurrent programs [15]. The availability of a refinement calculus provides us with the possibility of correctly constructing programs in a stepwise fashion. Hence, using *Circus* we are able to calculate concrete (usually distributed) specifications from abstract (usually centralised) specifications. Each step is justified by the application of a refinement law (possibly with the discharge of proof obligations). Together, the refinement laws provide us with a framework for the construction process. This derives from the fact that only valid laws can be applied at a certain time.

The manual application of the refinement calculus, however, is an error-prone and hard task; two well-known problems are the application of laws to large programs and the management of the proof obligations. In this paper, we present CRefine, a tool that supports the use of the *Circus* refinement calculus. It is freely available from `http://www.cs.york.ac.uk/circus`. Our initial intention is to use this tool to teach *Circus* and its refinement calculus.

CRefine was implemented in Java. Its user interface and architecture are similar to those of Refine [20], a tool that supports Morgan's refinement calculus [14] for sequential programs. Our tool is based on an early prototype [26] that was built to check the applicability of the *Circus* type checker. Nevertheless, we have considerably changed and extended this prototype. First, we have changed the *Circus* parser to a newer version that fixes a couple of bugs of its earlier version. This new parser is strongly based on the CZT framework [12], an initiative that started in 2002 that is geared towards providing extensible tool support for Standard Z in Java. By using this new parser, we are able to adapt CRefine to the architecture proposed for *Circus* tools in [6]. Furthermore, we are also able to extend the CZT Rules package, which provides classes for transforming and proving Z specifications, to *Circus*. This extended package is currently the gear used to transform *Circus* specifications in CRefine; this paper describes this process.

Besides changing CRefine internally, we also added new

facilities to manage developments: undoing and redoing refinement steps, saving and opening developments is now available. Furthermore, some GUI facilities like pretty-printing, filtering applicable laws according to the selected program, classification of laws, adding comments to the development, and printing the development were also included. In CRefine's current version, the discharge of some proof obligations is done automatically. The remaining proof obligations need to be manually discharged by the user, who can annotate them with the result of the proof within CRefine. All these facilities are described in this paper.

In the next section we present *Circus* and its refinement calculus. Section 3 describes CRefine's functionalities using a case study, its architecture and integration with the CZT. We draw some conclusions and discuss future work in Section 4.

2. *Circus* and its Refinement Calculus

Circus is a language that is suitable for the specification of concurrent and reactive systems; it has a theory of refinement associated to it. Its objective is to give a sound basis for the development of concurrent and distributed system in a calculational style like that of Morgan [14].

Circus programs are formed by a sequence of paragraphs, which can either be a Z paragraph, a declaration of channels, a channel set declaration, or a process declaration. In this section, we illustrate the main constructs of *Circus* using the specification of a simple chronometer (Figure 1). Both its components are initialised with zero; it can increment its time and output the current time.

All the channels must be declared; we give their names and the types of the values they can communicate. If a channel is used only for synchronisation, its declaration contains only its name. For example, in Figure 1, the stateless process *Clock* repeatedly communicates through channel *tick*.

The declaration of a process is composed by its name and specification. A process may be explicitly defined or defined in terms of other processes (compound). An explicit process definition contains a sequence of process paragraphs and a distinguished nameless main action, which defines its behaviour. We use Z to define the state. For instance, *AState* describes the state of the process *Chrono* that represents a chronometer: it contains the current *sec*ond and *min*ute stored in the chronometer.

Process paragraphs include Z paragraphs and declarations of (parametrised) actions. An action can be a schema, a guarded command, an invocation to another action, or a combination of these constructs using CSP operators.

The primitive action *Skip* does not communicate any value or changes the state: it terminates immediately. The action *Stop* deadlocks, and *Chaos* diverges; the only guarantee in both cases is that the state invariant is maintained.

$$
\begin{array}{l}
RANGE == 0 \mathinner{..} 59 \\
\textbf{channel}\ tick, time \\
\textbf{channel}\ out : RANGE \times RANGE \\
\\
\textbf{process}\ Chrono \mathrel{\widehat=} \\
\quad \textbf{begin state}\ AState == [\, sec, min : RANGE \,] \\
\quad\quad AInit == [\, AState' \mid sec' = min' \land min' = 0 \,] \\
\quad\quad IncSec == [\, \Delta AState \mid sec' = (sec + 1) mod\ 60 \\
\quad\quad\quad\quad\quad\quad \land\ min' = min \,] \\
\quad\quad IncMin == [\, \Delta AState \mid min' = (min + 1) mod\ 60 \\
\quad\quad\quad\quad\quad\quad \land\ sec' = sec \,] \\
\quad\quad Run \mathrel{\widehat=} (tick \rightarrow IncSec;\ ((sec = 0)\ \&\ IncMin) \\
\quad\quad\quad\quad\quad\quad \Box\ ((sec \neq 0)\ \&\ Skip))) \\
\quad\quad\quad\quad \Box\ (time \rightarrow out\,!(min, sec) \rightarrow Skip) \\
\quad\quad \bullet\ (AInit;\ (\mu\ X \bullet (Run;\ X))) \\
\quad \textbf{end} \\
\textbf{process}\ Clock \mathrel{\widehat=} \textbf{begin}\ \bullet \mu\ X \bullet tick \rightarrow X\ \textbf{end} \\
\textbf{process}\ TChrono \mathrel{\widehat=} (Chrono\ [\![\{\!|\ tick\ |\!\}]\!]\ Clock) \setminus \{\!|\ tick\ |\!\}
\end{array}
$$

Figure 1. Abstract Chronometer

The prefixing operator is standard, but a guard construction is also available.

$$p\ \&\ c?x \rightarrow A$$

For instance, if the condition p is *true*, the action above inputs a value through channel c and assigns it to x, and then behaves like A, which has the variable x in scope. If, however, p is *false*, the action blocks.

The CSP operators of sequence, external and internal choice, parallelism, interleaving, hiding may also be used to compose actions. Communications and recursive definitions are also available. *Chrono* has a recursive behaviour: after its initialisation, it recursively behaves like action *Run*, which represents the execution of a cycle of the chronometer. If it receives the indication that a second has passed, it increments the seconds and, if the seconds were set back to zero, it increments the minutes. However, if *Chrono* is asked to output its current time, it does so via channel *out*.

Seconds are incremented by one in the *IncSec* operation; however, every sixty seconds the seconds are set to zero, since the chronometer will start counting another minute. The Z operation *IncMin* increments the minutes. For simplicity, we consider that our chronometer counts only seconds and minutes. As for the seconds, the chronometer's minutes value resets every sixty minutes.

The parallelism and interleaving operators are different from those of CSP. We must declare a synchronisation channel set (in parallelism) and, to avoid conflicts, two sets that partition the variables in scope: state components, and input and local variables.

$$A_1\ [\![\, ns_1 \mid cs \mid ns_2 \,]\!]\ A_2$$

In the action above, the actions A_1 and A_2 synchronise on

the channels in the set cs. Both A_1 and A_2 have access to the initial values of all variables in ns_1 and ns_2, but A_1 may modify only the values of the variables in ns_1, and A_2, the values of the variables in ns_2.

References to parametrised actions need to be instantiated. Actions may also be defined using assignment, guarded alternation, or variable blocks. Finally, in the interest of supporting a calculational approach to development, an action can be a Morgan's specification statement [14].

The CSP operators of sequence, external and internal choice, parallelism, interleaving, and hiding may also be used to compose processes. Furthermore, the renaming $Poldc : newc$ replaces all the references to channels *oldc* by the corresponding channels in *newc*, which are implicitly declared. Parametrised processes may also be instantiated.

The process *TChrono* is the parallel composition of *Chronometer* and *Clock*. They synchronise on channel *tick*, which is hidden from the environment: iterations with *TChrono* can only be made through *time* and *out*. Some other operators are available in *Circus*, but are omitted here for conciseness.

2.1. Refinement in *Circus*

In [15], we describe a refinement strategy for *Circus*. From an abstract centralised specification, the strategy yields a distributed implementation. It is an interactive strategy, in which each iteration decomposes one process. An iteration typically includes three steps: a simulation that replaces the state components of the single abstract process with the components of all the distributed processes to be derived; action refinement that partitions the concrete state and actions in such a way that actions from one partition access only its components; and a process refinement that transforms the partitions in individual processes.

In the development of the chronometer, we distribute the minutes and seconds in two different processes: *Seconds* and *Minutes*. The former is responsible for the communication with the environment and for the seconds, and the latter is responsible for the minutes (Figure 2). They communicate via three internal channels: *Seconds* indicates to *Minutes* that it must increment its minutes via *inc*. It may also request the current minutes via *minsReq*; the answer is given via *ans*. The set of channels *Sync* groups these internal channels.

The refinement of the chronometer can be accomplished in a single refinement iteration. Since we do not intend to change the representation of minutes and seconds, this iteration involves no data refinement. In the action refinement, we split the *Chrono*'s state into two partitions: one that contains the *sec*onds and the other one that contains the *min*utes. The intention is to split the actions of process *Chrono* into two partitions as well: one interacts with the environment and contains the *sec*onds, and the other one contains the

$$
\begin{array}{l}
\textbf{channel}\ inc, minsReq \\
\textbf{channel}\ ans : RANGE \\
\textbf{chanset}\ Sync \mathrel{\widehat=} \{\!|\ inc, minsReq, ans\ |\!\} \\
\textbf{process}\ Seconds \mathrel{\widehat=} \\
\quad \textbf{begin state}\ SecSt \mathrel{\widehat=} [\, sec : RANGE \,] \\
\quad SecInit \mathrel{\widehat=} [\, SecSt' \mid sec' = 0 \,] \\
\quad IncSec \mathrel{\widehat=} [\, \Delta SecSt;\ \Xi MinSt \\
\qquad\qquad \mid sec' = (sec + 1) \bmod 60 \,] \\
\quad RunSec \mathrel{\widehat=} tick \rightarrow IncSec;\ (sec = 0)\ \&\ inc \rightarrow Skip \\
\qquad\qquad\qquad\qquad \Box\ (sec \neq 0)\ \&\ Skip \\
\qquad\qquad \Box\ time \rightarrow minsReq \rightarrow ans?mins \rightarrow \\
\qquad\qquad\qquad out!(mins, sec) \rightarrow Skip \\
\quad \bullet\ SecInit; (\mu\, X \bullet RunSec;\ X)) \\
\quad \textbf{end} \\
\textbf{process}\ Minutes \mathrel{\widehat=} \\
\qquad \textbf{begin state}\ MinSt \mathrel{\widehat=} [\, min : RANGE \,] \\
\qquad MinInit \mathrel{\widehat=} [\, MinSt' \mid min' = 0 \,] \\
\qquad IncMin \mathrel{\widehat=} [\, \Delta MinSt;\ \Xi SecSt \\
\qquad\qquad\qquad \mid min' = (min + 1) \bmod 60 \,] \\
\qquad RunMin \mathrel{\widehat=} inc \rightarrow IncMin \\
\qquad\qquad\qquad \Box\ minsReq \rightarrow ans!min \rightarrow Skip \\
\qquad \bullet\ MinInit; (\mu\, X \bullet RunMin;\ X) \\
\qquad \textbf{end} \\
\textbf{process}\ Chrono \mathrel{\widehat=} Minutes\ [\![Sync]\!]\ Seconds \setminus Sync
\end{array}
$$

Figure 2. Concrete Chronometer

*min*utes.

First, we introduce the paragraphs that belong to each individual partition into *Chrono*. The resulting process contains all the paragraphs of *Chrono*, *Seconds* and *Minutes*. One group of paragraphs accesses only *sec*, which is initialised to zero. Seconds are incremented using *IncSec*. The action *RunSec* represents a cycle in *Seconds*. If it receives the indication that a second has passed, it increments the seconds and, if the seconds were set back to zero, it indicates to the *Minutes* using *inc*. However, if it is asked to output the *time*, it asks the *Minutes* the number of minutes, receives the answer, and outputs the time via *out*. The other group of paragraphs accesses only *min*, which is also initialised to zero. Minutes are incremented using *IncMin*. The action *RunMin* represents a cycle in *Minutes*. If it receives a request to *inc*rement the minutes, it does so. However, if the number of minutes is requested via *minsReq*, it outputs the value of *min* in *ans*.

The next refinement steps are achieved using the Z schema calculus [25] and the copy rule; they yield the declaration below as the new state declaration for process *Chrono*.

$$\textbf{state}\ State \mathrel{\widehat=} SecSt \wedge MinSt$$

Next, we need to transform the main action of the *Chrono* into a parallel composition of the behaviour of each partition; this transformation is summarised in Figure 3. For conciseness, in the remaining of this section we represent the parallel composition operator as parallel bars; hence, omitting

the declaration of the state partitions and the synchronisation channel set.

$$
\begin{array}{l}
AInit; (\mu\, X \bullet Run;\ X) \\
= MinInit;\ SecInit; (\mu\, X \bullet Run;\ X) \qquad \text{[Law A.1]} \\
\sqsubseteq_{\mathcal{A}} \qquad \text{[Law A.2]} \\
MinInit;\ SecInit; \\
((\mu\, X \bullet RunSec;\ X) \parallel (\mu\, X \bullet RunMin;\ X)) \setminus Sync \\
= \qquad \text{[Laws A.3 and A.4]} \\
\left(\begin{array}{l} MinInit;\ SecInit; \\ (\mu\, X \bullet RunSec;\ X) \parallel (\mu\, X \bullet RunMin;\ X) \end{array} \right) \\
\setminus Sync \\
= \qquad \text{[Law A.5]} \\
\left(\begin{array}{l} (SecInit; (\mu\, X \bullet RunSec;\ X)) \\ \parallel (MinInit;\ (\mu\, X \bullet RunMin;\ X)) \end{array} \right) \\
\setminus Sync
\end{array}
$$

Figure 3. Proof of Refinement

The process of refining actions consists of repeatedly applying refinement laws until we reach the desired concrete action. As an example, we have the Law A.1 presented in Appendix A that splits an initialisation operation into a sequence. This law applies to a schema S which operates over a state composed of two disjoint partitions S_1' and S_2'. The updates of S on the state are expressed as a conjunction of two predicates P_1 and P_2, whose free-variables are components of S_1' and S_2', respectively. It transforms the given schema into a sequence of two schemas; each of them corresponds to the original operation on one of the state partitions.

The initialisation *AInit* meets all the provisos of the Law A.1. Hence, using the copy-rule and this law, we may transform it into a sequence of two different initialisations: *MinInit* and *SecInit*. For conciseness, we omit all proof obligations and informally justify the refinement steps.

Next, we use the least fixed-point law (A.2) to split the recursion into a parallel composition of two recursions that correspond to the behaviour of the *Seconds* and *Minutes*, respectively; this is justified by Lemma 2.1, which is stronger that the law proviso. Next, since schema expressions use no channels, we may expand the hiding. Finally, the schemas change only variables declared in one of the partitions of the parallel composition, and the variables they write to are not used by the other side of the composition. For this reason, we may move each of them to one of the sides of the composition. This concludes the proof of the action refinement. However, we are still left with the proof of the Lemma 2.1; it can be found elsewhere [15].

Lemma 2.1

$$
\begin{array}{l}
((\mu\, X \bullet RunSec;\ X) \parallel (\mu\, X \bullet RunMin;\ X)) \setminus Sync \\
\sqsubseteq_{\mathcal{A}} \\
Run;\ ((\mu\, X \bullet RunSec;\ X) \parallel (\mu\, X \bullet RunMin;\ X)) \setminus Sync
\end{array}
$$

Informally, we start from the left-hand side, unfolding the first recursion. Afterwards, we distribute the recursion through each of the choices that are in the recursion body. Then, we combine the second recursive program in parallel with each of the branches of the first recursion. The strategy is to show that each of these branches can be transformed into a branch of *Run* followed by the left-hand side itself. This yields a program which coincides with the body of the recursion on the right-hand side, except that in place of the recursive call we have the left-hand side itself. The distribution laws and the definition of action *Run* concludes this proof.

We are left with the concrete main action: both partitions initialise their state components and execute their cycles recursively. They synchronise on *Sync* that is hidden from the environment.

After this action refinement, we are allowed to remove the paragraphs of the original *Chrono* since they are no longer referenced within this process. Finally, we have a process with a state partitioned into two: one is concerned with the seconds and the other one is concerned with the minutes. Each partition has its own set of paragraphs, which are disjoint, since no action in one changes a state component in the other. The main action of the refined process is defined in terms of the parallel composition of actions from both partitions. The final step of our refinement uses the process refinement Law A.6 in order to rewrite the process *Chrono* in terms of two independent processes, *Seconds* and *Minutes*, as presented in Figure 2.

Using a tool like CRefine, we can mechanise the whole development of the chronometer presented here. It is the subject of the next section.

3. CRefine

CRefine automatises the application of the *Circus* refinement calculus: it supports the application of refinement laws to *Circus* specifications and the management of the development. In Section 3.1 we describe CRefine's functionalities and illustrate them using the example presented in Section 2.1. Due to space limitations, we are not able to describe all these functionalities in details; they can be found in the tool distribution on the web. Finally, CRefine's internal structure and its integration with the CZT framework is described in Section 3.2.

3.1. Using CRefine

CRefine's GUI is composed by a main menu and three main frames: refinement, proof obligations, and code; they are illustrated in Fig. 4. The refinement frame shows all the steps of the development. This includes law applications and

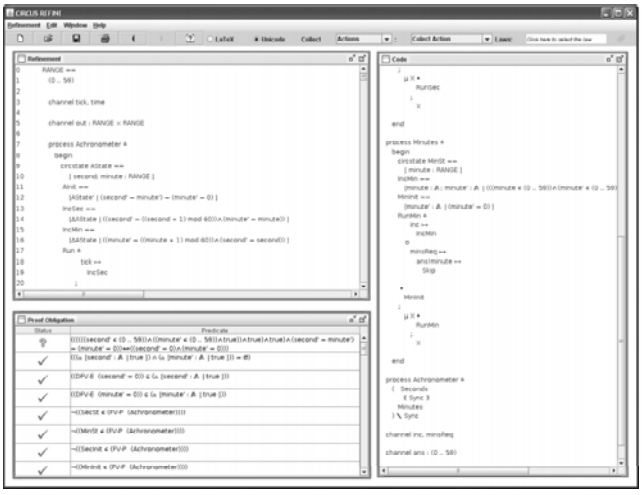

Figure 4. CRefine User Interface

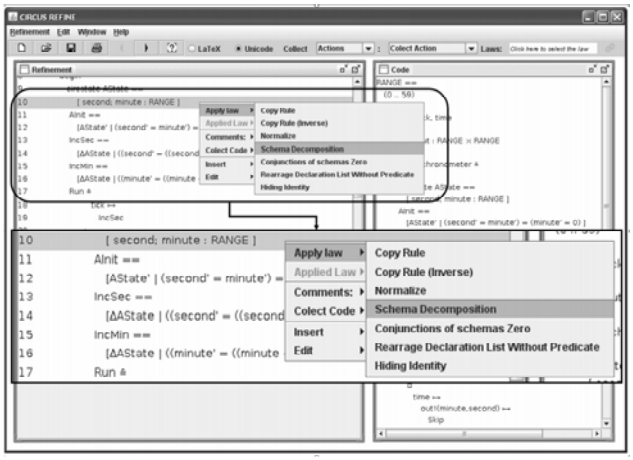

Figure 5. Applying a Refinement Law

retrieving the current status of an action or process (collection). The proof obligations frame lists the proof obligations that are generated by law applications, indicates their current state (i.e. checked valid or invalid, or unchecked), and associates each proof obligation to the law application that originated it in the refinement frame. Currently, some proof obligations are automatically verified. In our experience, these amount to over 65% of them. The remaining ones need to be verified by the user; the result can be registered in CRefine. Finally, the code frame exhibits the whole specification that has been calculate so far.

CRefine provides two display formats for formulas: LaTeX and Unicode (pretty-printing). Our main objective is to use this tool in teaching the *Circus* refinement calculus to under-graduates and, unfortunately, most of them are not familiar with LaTeX. Hence, a pretty-printing makes CRefine accessible to them. This pretty-printing is also a success among researchers, since it unconditionally makes the presentation of the development more user-friendly.

In CRefine, a development starts by loading a LaTeX document that contains the abstract specification of the system. This can be achieved by clicking on the *new* button and selecting the input file. Thereafter, the application of refinement laws is as follows: first, we select the *Circus* term that we want to refine. A pre-formatting allows us to select terms on a per line basis by clicking on its lines (multiple lines can be selected by holding `ctrl` and clicking on the first and last line of the term). Next, we select the law we want to apply. Finally, after the input of any arguments that may be required by the law, the application is automatically done. This updates the refinement frame, the proof obligations frame (if needed), and the code frame.

Law selection and application can be achieved in two ways. First, a right-click on the selected term shows a pop-up menu that lists only those laws that can be applied to the selected term. In this case, their effective application is done by selecting them in this list. Second, we can select the law from a list in the main menu that contains all the refinement laws. If the law can be applied to the term, the apply button is enabled and can be pushed to effectively apply the law. In Figure 5, we have a snapshot of the development of our example within CRefine in which point we intend to split the global *State* into a conjunction of the two individual states *SecSt* and *MinSt* (the screenshots that follows zoom on items that are discussed in the text). Among the laws we can apply to the definition of *AState* (the selected term), we have the *Schema Decomposition* law. A click on the *Schema Decomposition* item applies this law to the selected term. An argument window may be displayed during law applications. Using it, the user inputs the arguments needed. If the Unicode format is being used, the user can use the virtual symbol keyboard provided (Figure 6) to input *Circus* symbols in the argument window. The user can also check the details of any refinement law.

In our example, we give the name of the component of the first schema, *sec*, as argument; the law application yields the following conjunction.

Each law application has an identifier that associates the term to which the law was applied with the final result of this application. The *Schema Decomposition* has no proof obligations. Hence, the proof obligations frame remains unchanged. Nevertheless, the code window is changed: the definition of the *AState* is now the conjunction described before as we can see in Figure 7. For the sake of conciseness, we have illustrated just some points of the whole development of our example within CRefine, which amounts to over 1000 lines and can be found within the distribution of the tool.

Using CRefine, one cannot apply refinement laws to

Figure 6. Virtual Symbol Keyboard

$$[sec : RANGE] \land [min : RANGE]$$

the same term twice. For instance, after the application of *Schema Decomposition*, we have changed the whole body of *Chrono* (part between **begin** and **end**). Hence, its selection disables the *apply law* item. Law applications to individual paragraphs and actions of process *Chrono*, however, is still possible without collecting the code; only those terms that have been changed, like the whole process, need to be collected before further law applications to them.

Some refinement law applications are justified by proof obligations; they are displayed in the proof obligation frame. CRefine keeps the relation of each proof obligation to the refinement step that generated it. For instance, in Figure 8, CRefine is relating a proof obligation to the corresponding application of Law A.1 during the development of our example. In this case, CRefine has managed to automatically prove the proof obligation; this indicated with a green ✔ to the left of the proof obligation.

Using CRefine's development management, users may undo and redo development steps. That means different development paths during a development may be tried, possibly in a search for a more efficient implementation. Developments may also be saved (XML file) and opened later. Users may document the development by adding, editing, and viewing comments to each term or law application in the refinement window. Finally, CRefine generates LaTeX files that document the chosen elements of developments: original specification, refinement, proof obligations, comments and the concrete specification.

3.2. CRefine's Architecture

CRefine is designed towards an integration with the *Circus* tools presented in [6]. In Figure 9, we present a simplified view of the components that are part of CRefine and the interaction between some of them and the CZT components

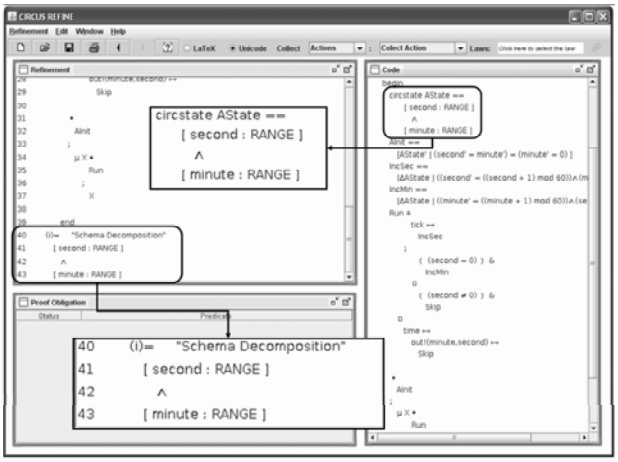

Figure 7. Result of Applying Refinement Laws

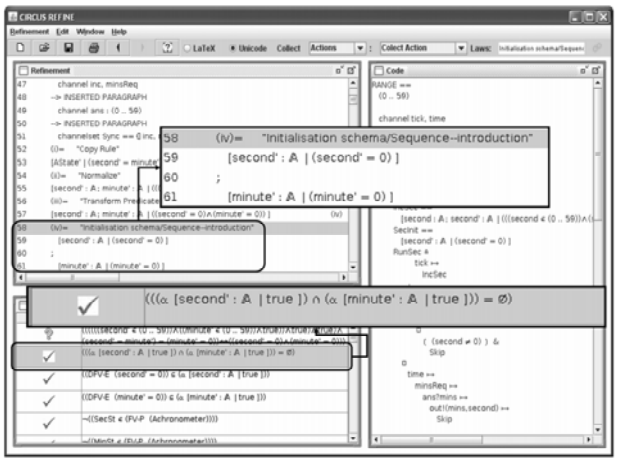

Figure 8. Proof Obligations

described in [6]. As we describe in the sequel, some of them are yet to be implemented; their names are in *italic* font.

CRefine's GUI controls the whole interaction with the user: it receives interactive commands, sends these commands to the External Manager (EM), and updates the frames accordingly. It interacts with the *Circus* printer developed by us in order to update the frames using the display format chosen by the user. Although the CZT already provides a pretty-printer for *Circus*, we needed to develop our own. In CRefine, users need to click on the lines of a term in order to select it; hence, the choice of the points of a *Circus* specification in which a line break is inserted by the pretty-printer has a direct impact in the type of terms that can be selected by the user. In the existing *Circus* pretty-printer, line breaks do not play such a major role, as in CRefine.

The EM connects to the GUI and the Internal Man-

ager (IM). It is responsible for controlling the relations between the lines in the development frame and the corresponding term. Furthermore, it relates the proof obligation to the law application in the development. Finally, it also interacts with the document generator to create a LaTeX file that documents the development.

The IM stores the status of the tool, the refinement laws that are loaded during CRefine's start-up using the Laws Factory, and manages the overall development. The status of CRefine contains the current Abstract Syntax Tree (AST), two stacks of ASTs for undoing and redoing, and the information needed to save the development: a list of steps of execution (actions done by the user like law applications and comments insertion). In the beginning of a new development, the IM interacts with the *Circus* Specification Processing Module (SPM) in order to transform a LaTeX specification into an AST. Currently, we are not using the Type Checker, which is being validated. In a near future, this type checker will be included in the SPM: the AST that will be given by the SPM to CRefine will contain some type annotations (denoted by AST+ in Figure 9). The IM also interacts with the SPM in order to parse arguments that are given by the user. Furthermore, it is responsible for filtering (using the *Circus* Rules module) the laws that can be applied to the selected term in the GUI. Finally, the application of the refinement laws is done through an interaction between the IM and *Circus* Rules, which actually applies the refinement law.

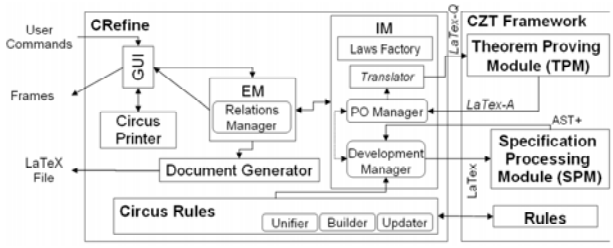

Figure 9. Integration of CRefine **and the** CZT

The current proof obligations manager (PO Manager) automatically discharge some proof obligations. In a near future, we intend to integrate this module with the Theorem Proving Module (TPM) of the CZT. For this, we will need a translator that will transform the proof obligation into a LaTeX question (LaTeX-Q) to be sent to the TPM. The PO manager will analyse the LaTeX answers (LaTeX-A) returned by the TPM. Besides integrating with the TPM, the PO manager will also need to be internally integrated with the developments manager. This will allow the mechanical proof of proof obligations like the Lemma 2.1, which we needed to prove in order to validate the development of our example in Section 2.1.

Each module presented in Figure 9 has a precise role in the overall development. In Figure 10, we present a simplified sequence diagram that describes a law application. In this diagram, we replace method signatures by the task name. We can split a law application in four stages: selecting a term, finding applicable laws, selecting a law, and applying it (possibly inputting arguments).

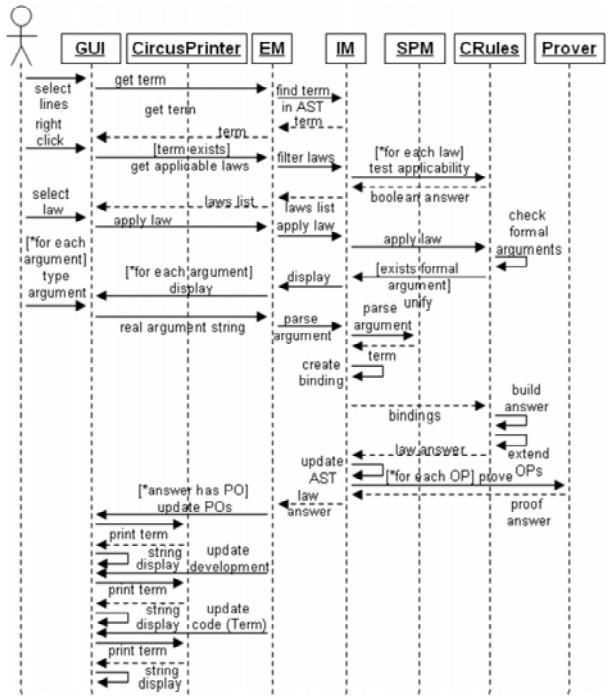

Figure 10. Diagram - applying a Refinement Law

For every line selection in the development window, the GUI request to the EM to return the term that corresponds to the selected lines. Having selected a term, every right click in the development window will update the list of laws that are applicable to the selected term. First, the GUI requests the list of applicable laws to the EM. This request is propagated to the IM, who uses the Unifier of the *Circus* Rules in order to return only those laws that unify with the received term. This list of laws is given to the GUI and exhibited to the user.

The selection of the term and of the law results in a law application. It is propagated from the GUI to the IM who sends the original term and the law to the *Circus* Rules that checks if the application requires arguments. If no arguments are required, the application takes part. Otherwise, the *Circus* rules sends to the EM (via the IM) a list of formal arguments. For each one of them, the EM interacts with the GUI in order to receive a string that corresponds to the real argument. It is given to the SPM (via the IM) that parses it and returns a term. Next, the IM maps names to terms (bindings) and sends a list of bindings to the *Circus*

rules that is now able to apply the law. The *Circus* rules unifies the law with the original term (Unifier), builds a new term and proof obligations (Builder), and updates the AST accordingly (Updater). The answer that is returned to the IM contains, among other information, the new term and a list of proof obligations. This is used by the IM to update its own AST. Next, the IM uses the prover to verify each proof obligation: if the prover can handle the proof obligation, the result is either *false* or *true*; otherwise, the proof obligation remains unverified. This concludes the task of the IM that returns the law answer to the EM. Finally, the EM updates all the frames. Every update in the GUI interacts with the *Circus* printer: the GUI sends the term to the printer and receives the corresponding string in the current display format.

4. Conclusions

In this paper, we presented CRefine, a tool that supports stepwise development of state-rich concurrent systems based on the *Circus* refinement calculus. It extends an earlier prototype by adding new features like new interface components and development management facilities. Using CRefine, we considerably reduce the amount of effort spent in the application of the *Circus* refinement calculus: tasks like law applications and proof obligation generations are automated by the tool.

The design of CRefine was driven by our intention to integrate CRefine with existing *Circus* tools like its parser, type-checker and theorem prover, which are part of the CZT: we have already achieved its integration with the parser and an integration with the type-checker is under way. We have extended the CZT rules package in order to apply refinement laws as rules transformations. We believe that, with this paper, we provide useful information to tool builders in how a refinement process can be managed in a GUI as well as a template for refinement tools internal architecture.

Several existing tools provide support for the use of the refinement calculus and tactics. Some [8, 3] use languages like Prolog for defining tactics and, in this case, the user needs to learn complex languages to achieve his goals. In [9] a goal-oriented approach is adopted: refining consists of proving that the final program implements the initial specification. The Proxac system [24] does not define any language for tactic definition. It is the support provided for refinement of state-rich reactive systems in a calculational style that makes CRefine a novelty.

Frequently used refinement strategies are reflected in sequences of laws applications. Identifying these strategies, documenting them as tactics, and using them as single transformation rules brings a profit in effort. In [16], we present a tactic language, ArcAngel*C* that can be used to formalise tactics of refinement just like in [17]. Allowing users to define and use tactics within CRefine will reduce the amount of effort needed during developments.

The majority of the refinement laws from [15], which have been used in a reasonable number of case studies, are in CRefine. This gives us confidence that the current set of laws is appropriate for useful applications, but not complete [15]. In order to facilitate extensions to the set of laws, we intend to provide a parser of refinement laws in the style of CZT. Using this parser, the laws could be dynamically loaded; no recompilation would be needed.

The automatic discharge of the proof obligations that are predicates or action transformations (i.e Lemma 2.1) is also in our Agenda. The former requires the integration of CRefine with a more elaborated theorem-prover and the latter requires the possibility of multiple developments within CRefine to discharge proof obligations that are action transformations.

Finally, the infra-structure provided by tactics of refinement and multiple developments can be used to allow users to make sub-developments within larger developments just like the window inference transformational style of [2]. This would considerably modularise future developments.

Using *Circus*, we are able to make a stepwise development in a single framework from a specification to an implementation. Nevertheless, we still need to translate this *Circus* implementation into a practical programming language because the final product of a program development is an executable program. An automatic translator has already been provided [5] and will be integrated to CRefine allowing a complete development within our tool.

CRefine can be a useful tool in the development of state-rich reactive systems. Initially, we intended to develop an educational tool and use it in teaching formal methods. However, during the project, we noticed that it may be useful in the development of real systems. So far, the tool has been used only by members of our research group on a small number of case studies. In the near future, we intend to use CRefine in our formal methods course. Besides, we are currently working on a number of case studies that are related to the oil industry.

5. Acknowledgments

ANP and CNPq supports the work of the authors: grant 550946/2007-1. We are grateful to Leo Freitas for valuable discussions about the CZT framework. Ana Cavalcanti and Manuela Xavier provided us with insights on the previous version of CRefine.

A. Refinement Laws

The side conditions of some of the refinement laws involve meta-functions such as α, DFV, $usedC$, $usedV$, and

wrtV. The function α determines the set of components of a given schema; for a given predicate p, the function *DFV* yields the dashed free variables of p; the function *usedC* returns a set of all channels mentioned in an action; the function *usedV* gives the set of used variables (read, but not written); finally, the function *wrtV* gives the set of variables that are written by a given action.

Law A.1 (Initialisation/Sequence-introduction)
$[S_1'; S_2' \mid P_1 \wedge P_2] = [S_1' \mid P_1]; [S_2' \mid P_2]$
provided

- $\alpha(S_1) \cap \alpha(S_2) = \emptyset$
- $DFV(CS_1) \subseteq \alpha(S_1')$
- $DFV(CS_2) \subseteq \alpha(S_2')$

Law A.2 (Recursion-least fixed-point)
$F(Y) \sqsubseteq_{\mathcal{A}} Y \Rightarrow \mu X \bullet F(X) \sqsubseteq_{\mathcal{A}} Y$

Law A.3 (Hiding Identity)
$A \setminus cs = A$
provided

- $cs \cap usedC(A) = \emptyset$

Law A.4 (Hiding/Sequence-distribution)
$(A_1; A_2) \setminus cs = (A_1 \setminus cs); (A_2 \setminus cs)$

Law A.5 (Schemas/Parallelism-distribution)
$SExp; (A_1 \,[\![ns_1 \mid cs \mid ns_2]\!]\, A_2)$
$=$
$(SExp; A_1) \,[\![ns_1 \mid cs \mid ns_2]\!]\, A_2$
provided

- $wrtV(SExp) \subseteq ns_1$
- $wrtV(SExp) \cap usedV(A_2) = \emptyset$

Law A.6 (Process splitting)
Let qd and rd stand for the declarations of the processes Q and R, determined by Q.State, Q.PPar, and Q.Act, and R.State, R.PPar, and R.Act, respectively, and pd stand for the process declaration.

process $P \mathrel{\widehat=}$
begin
state $State \mathrel{\widehat=} Q.State \wedge R.State$
$Q.PPar \wedge_{\Xi} R.State$
$R.PPar \wedge_{\Xi} Q.State$
$\bullet F(Q.Act, R.Act)$
end

then
$pd = (qd\ rd$ **process** $P \mathrel{\widehat=} F(Q, R))$
provided

- *Q.PPar and R.PPar are disjoint with respect to R.State and Q.State*

References

[1] J.-R. Abrial. *The B-book: Assigning Programs to Meanings*. Cambridge University Press, 1996.

[2] M. Butler, J. Grundy, T. Långbacka, R. Rukšėnas, and J. von Wright. The Refinement Calculator: Proof Support for Program Refinement. In L. Groves and S. Reeves, editors, *Formal Methods Pacific'97: Proceedings of FMP'97*, pages 40–61, Wellington, New Zealand, 1997. Springer-Verlag.

[3] D. Carrington, I. Hayes, R. Nickson, G. Watson, and J. Welsh. A program refinement tool. *Formal Aspects of Computing*, **10**(2):97–124, 1998.

[4] C. Fischer. CSP-OZ: A combination of Object-Z and CSP. In H. Bowman and J. Derrick, editors, *Formal Methods for Open Object-Based Distributed Systems (FMOODS'97)*, volume **2**, pages 423–438. Chapman & Hall, 1997.

[5] A. Freitas and A. L. C. Cavalcanti. Automatic Translation from *Circus* to Java. In J. Misra, T. Nipkow, and E. Sekerinski, editors, *14th International Symposium on Formal Methods*, volume **4085** of *Lecture Notes in Computer Science*, pages 115–130. Springer, Aug 2006.

[6] L. J. S. Freitas, J. C. P. Woodcock, and A. L. C. Cavalcanti. An Architecture for *Circus* Tools. In A. C. V. Melo and A. Moreira, editors, *Proceedings of the Brazilian Symposium on Formal Methods*, pages 6 – 21, 2007.

[7] The RAISE Language Group. *The RAISE Specification Language*. Prentice-Hall, 1992.

[8] L. Groves, R. Nickson, and M. Utting. A Tactic Driven Refinement Tool. In C. B. Jones, R. C. Shaw, and T. Denvir, editors, *5th Refinement Workshop*, Workshops in Computing, pages 272–297. Springer-Verlag, 1992.

[9] J. Grundy. A Window Inference Tool for Refinement. In C. B. Jones, R. C. Shaw, and T. Denvir, editors, *5th Refinement Workshop*, Workshops in Computing, pages 230–254. Springer-Verlag, 1992.

[10] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[11] C. A. R. Hoare and He Jifeng. *Unifying Theories of Programming*. Prentice-Hall, 1998.

[12] P. Malik and M. Utting. CZT: A Framework for Z Tools. In H. Treharne, S. King, M. C. Henson, and S. A. Schneider, editors, *ZB*, volume **3455** of *Lecture Notes in Computer Science*, pages 65–84. Springer, 2005.

[13] R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.

[14] C. Morgan. *Programming from Specifications*. Prentice-Hall, 1994.

[15] M. V. M. Oliveira. *Formal Derivation of State-Rich Reactive Programs using Circus*. PhD thesis, Department of Computer Science, University of York, 2006. YCST-2006/02.

[16] M. V. M. Oliveira and A. L. C. Cavalcanti. ArcAngelC: a Refinement Tactic Language for *Circus*. *Electronic Notes in Theoretical Computer Science*, **214C**:203 – 229, 2008.

[17] M. V. M. Oliveira, A. L. C. Cavalcanti, and J. C. P. Woodcock. ArcAngel: a Tactic Language for Refinement. *Formal Aspects of Computing*, **15**(1):28–47, 2003.

[18] M. V. M. Oliveira, A. L. C. Cavalcanti, and J. C. P. Woodcock. A UTP Semantics for *Circus*. *Formal Aspects of Computing*, 2008. DOI 10.1007/s00165-007-0052-5.

[19] M. V. M. Oliveira, A. L. C. Cavalcanti, and J. C. P. Woodcock. Unifying Theories in ProofPower-Z. *Formal Aspects of Computing*, 2008. DOI 10.1007/s00165-007-0044-5.

[20] M. V. M. Oliveira, M. Xavier, and A. L. C. Cavalcanti. Refine and Gabriel: Support for Refinement and Tactics. In Jorge R. Cuellar and Zhiming Liu, editors, *2nd IEEE International Conference on Software Engineering and Formal Methods*, pages 310–319. IEEE Computer Society Press, Sep 2004.

[21] N. Plat and P. G. Larsen. An Overview of the ISO/VDM-SL Standard. *ACM SIGPLAN Notices*, **27**(8):76–82, 1992.

[22] A. W. Roscoe, J. C. P. Woodcock, and L. Wulf. Noninterference through Determinism. In D. Gollmann, editor, *ESORICS 94*, volume **875** of *Lecture Notes in Computer Science*, pages 33–54. Springer-Verlag, 1994.

[23] K. Taguchi and K. Araki. The state-based CCS semantics for concurrent Z specification. In M. Hinchey and Shaoying Liu, editors, *International Conference on Formal Engineering Methods*, pages 283–292. IEEE, 1997.

[24] J. L. A. van de Snepscheut. Proxac: An Editor for Program Transformation. Technical Report 1993.cs-tr-93-33, 1993.

[25] J. C. P. Woodcock and J. Davies. *Using Z—Specification, Refinement, and Proof*. Prentice-Hall, 1996.

[26] M. A. Xavier, A. L. C. Cavalcanti, and A. C. A. Sampaio. Type Checking *Circus* Specifications. In A. M. Moreira and L. Ribeiro, editors, *SBMF 2006: Brazilian Symposium on Formal Methods*, pages 105 – 120, 2006.