Modelling and analysis of real-time coordination patterns

Kemper, S.

Chapter 5

Tool Development and Application to Case Studies

In the previous Chapters, we have presented the theoretical foundations of a framework for modelling and analysing distributed real-time systems. While it is of course very important to have a well-grounded and correct theoretical basis to start from, any formalism can only be used in practice if it comes along with adequate tool support. Such tool support can in turn be used to show the usability and applicability of a formalism. In particular, graphical editors offer intuitive and easy access to a new formalism, even for the unexperienced user. In this Chapter, we present our work on tool development, which has been done in the context of this thesis.

The rest of this chapter is organised as follows: in Section 5.1, we present the details of the tool implementation. We start by giving a brief introduction to the *Extensible Coordination Tools* (ECT) in Section 5.1.1, a tool suite that is being developed in the Foundations of Software Engineering group[1] (SEN3) at CWI. In the remainder of Section 5.1, We then present the support for modelling and analysis of TCA that we have integrated into ECT . In Section 5.2, we use a small example to explain the most important features of our tool, and show the typical workflow when using it. Finally, in Section 5.3, we introduce two case studies, present experimental results when using our tool to model check them, and discuss the advantages of using our formalism and tool over using other formalisms.

Except for Sections 5.1.1 (which is presented to provide a deep insight into the functioning of our implementation in the overall context of ECT) and 5.3.1 (which has been presented in [Kem11]), this Chapter describes original work, which has not been presented elsewhere.

[1] `http://www.cwi.nl/research-groups/Foundations-of-software-engineering`

5.1 Implementation

We have implemented our work on TCA and integrated it as part of the *Extensible Coordination Tools* (ECT, [ECT]), building on the *Extensible Automata (EA) framework*. In the next section (Section 5.1.1), we give a high-level overview of ECT, and briefly introduce the architecture of the EA framework. In Section 5.1.2, we present the implementation of the TCA editor plugin. Finally, Section 5.1.3 explains the implementation of the formula generation from TCA. If not explicitly mentioned, all implementation is done in *Java* [GJSB05].

5.1.1 The Extensible Automata framework in ECT

ECT is an integrated graphical development environment available for the Eclipse [Ecl] platform. It consists of a set of plugins that support modelling and analysis of component-based systems. The core of ECT has been developed and implemented in the context of the PhD thesis of Christian Krause [Kra11]. ECT provides extensive support for systems which are specified in the channel-based coordination language $\mathcal{R}$eo [Arb04], including a graphical modelling environment (editor), conversion to and from other models (for example to quantitative intentional automata (QIA) and (in turn) markov chains [AMM+09], to CA [ABRS04], to mCRL2 [KKdV10, Kra11], from BPMN, UML sequence diagrams and BPEL [CKA10]), java code generation, and animation. In addition, ECT comprises plugins to directly edit most of the aforementioned models, amongst other for CA, QIA and mCRL2 (for BPMN, an Eclipse plugin outside ECT already exists [BPM]). Please refer to [Kra11, ECT] for a complete and detailed description of ECT.

Our implementation of the TCA plugin uses the *Extensible Automata* (EA) framework in ECT. The framework was developed and implemented with the intention to "provide a unified framework for deriving automata-based models for $\mathcal{R}$eo" [Kra11], but allows to extend existing and define new automata-based models in general (i.e., detached from $\mathcal{R}$eo). In the remainder of this Section, we give a brief overview of the EA framework; again, we refer to [Kra11] for a complete and detailed description.

The meta model of the framework comprises the packages *cwi.ea.automata* (cf. Figure 5.1) and *cwi.ea.extensions* (cf. Figure 5.2).[2] Conceptually, there are two different types of elements in the EA framework: extensible elements and extensions (extending elements). The interfaces *IExtensible* in *cwi.ea.automata* and *IExtension* in *cwi.ea.extensions* mirror this structure. Every *IExtensible* owns a number of *IExtension*s.

Package *cwi.ea.automata* contains classes for the basic extensible elements that make up every automata-based model: Automaton, State,[3] and Transition. These extend the abstract base class *ExtensibleElement* (in package *cwi.ea.extensions*), which implements interface *IExtensible*.

[2] Figures 5.1 and 5.2 are essentially taken from [Kra11]. Equivalent diagrams can also be found as part of the source code of the ECT tools, publicly available at `http://reo.project.cwi.nl/`.

[3] Our notion of *location* (cf. Chapter 2) equates to the notion of *state* in the EA framework.

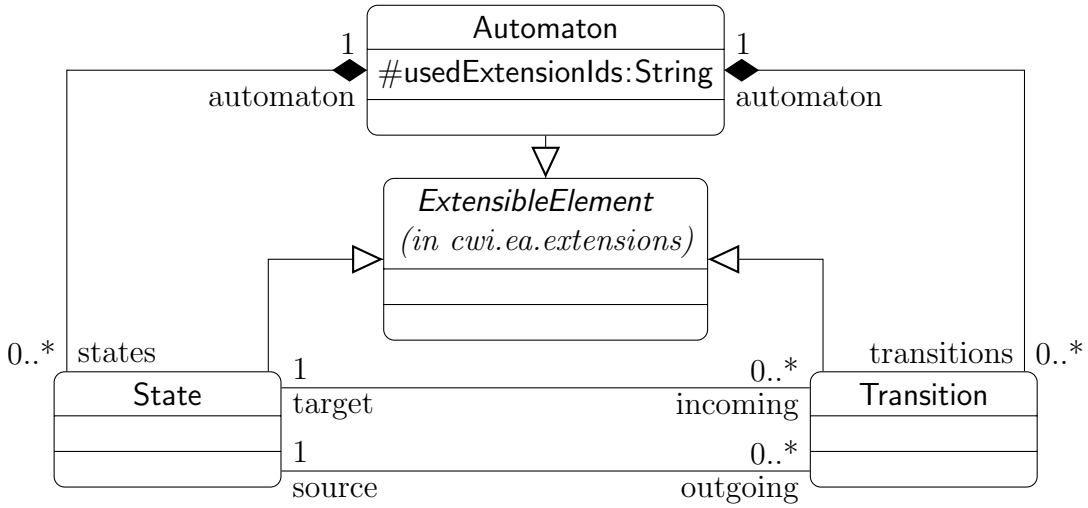

Figure 5.1: Package *cwi.ea.automata*

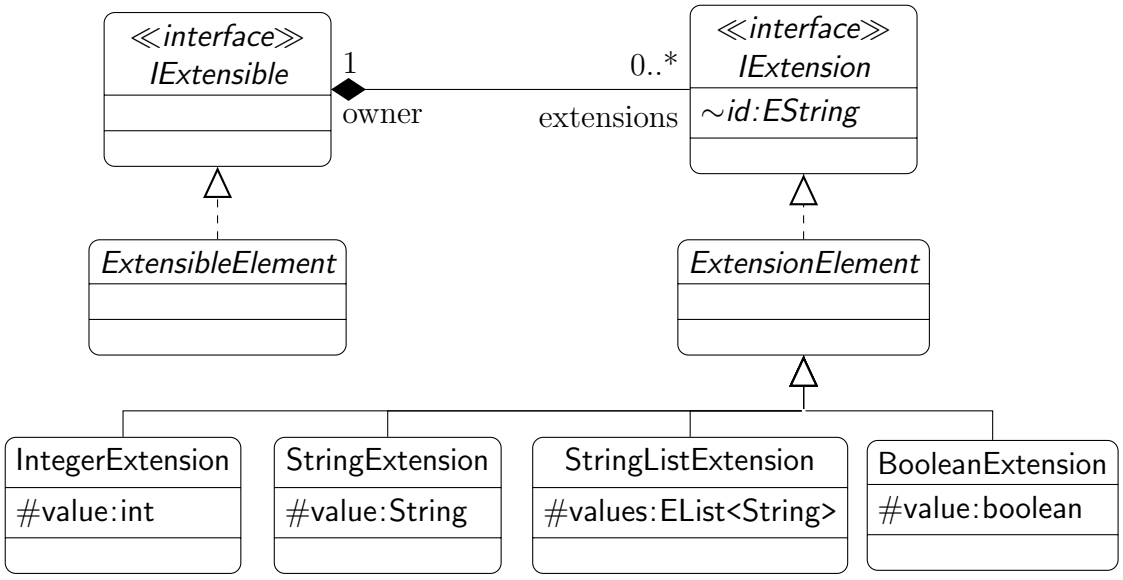

Figure 5.2: Package *cwi.ea.extensions*

Extensions are key/value pairs, where the key is a unique ID of type String, and the value contains the content information of the extension. There are four predefined extension value types in *cwi.ea.extensions*: StringExtension, StringListExtension, BooleanExtension and IntegerExtension. These extend the abstract base class *ExtensionElement*, which implements *IExtension*. New custom types can be easily defined by extending *ExtensionElement* or one of its subclasses.

To make clear the difference between *plugin* and *extension*: a plugin is an encapsulation of behaviour, providing support for a certain feature. For example, the TCA plugin provides support for modelling and analysing TCA in various ways. A plugin typically comprises several extensions. Every extension implements one aspect of the feature, for example, modelling TCA transitions by adding real-time aspects to transitions. Every extension is enabled for (i.e., applicable to) exactly one of the extensible elements Automaton, State or Transition.

Every extension has a provider class (*extension provider*), cf. Figure 5.3. An extension provider has to implement the interface *IExtensionProvider*, and one of the interfaces *ITextualExtensionProvider* (for textual extensions, i.e. labels) or *ICustomExtensionProvider* (for other types of extensions), all contained in package *cwi.ea*. The provider class defines how to handle the extension in the editor, it contains methods amongst others for parsing, editing and validating (syntax and semantic checks) the extension, or for providing a default extension. For conciseness of explanation, in the sequel, we refer to extensions by the name of their provider class, while omitting the suffix Provider.

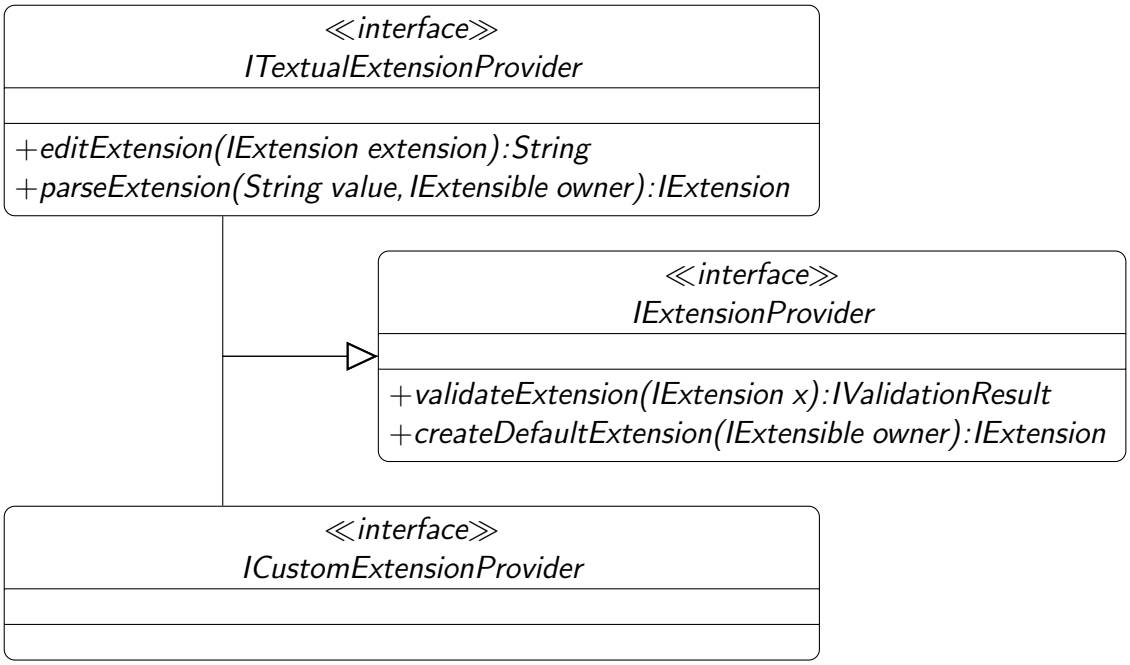

Figure 5.3: Package *cwi.ea*

An EA automaton can in principle use any combination of the extensions defined in plugins within the EA framework (see [Kra11] for a complete overview of supported extensions). In the manifest file `plugin.xml` that accompanies every plu-

gin definition in Eclipse, it is possible to define dependencies and mutual exclusion constraints among extensions (either among extensions of the particular plugin, or among extensions of the particular plugin and other plugins). Moreover, in the `plugin.xml` files of EA plugins, it is possible to define an automaton type, that has a predefined set of extensions enabled on creation.

5.1.2 The Timed Constraint Automaton Plugin

We have added support for TCA to the EA framework. Since TCA are an extension of CA, we were able to use the existing extensions for initial locations (StartStateExtension), port names on automaton level (AutomatonPortNames), active ports (TransitionPortNames), and location memory (StateMemoryExtension). To support the particular features of TCA, we implemented a number of new extensions. All of the new extensions implement interface *ITextualExtensionProvider* (cf. Figure 5.3), and are simple enough such that we were able to use the predefined extension value type StringExtension (cf. Figure 5.2). All extensions are contained in package *cwi.ea.extensions.clocks*. The following overview shows the new extensions, their format as seen in the editor, and their most important features. The default value is generated when calling *createDefaultExtension()* , the syntactic checks are performed when calling *validateExtension* (both from *IExtensionProvider*).

AutomatonClocks (all clocks defined for a TCA): comma separated list of clock names. Enabled for Automaton.[4] Default value "" (empty string, i.e., no clocks).

StateInvariant (location invariants): clock constraint formula according to Definition 2.1.2. Enabled for State. Default value `true`. Syntactic check that formula is well-formed, and that every clock used in the formula is defined in AutomatonClocks.

TransitionGuard (clock guards on transitions): clock constraint formula according to Definition 2.1.2. Enabled for Transition. Default value `true`. Syntactic check that formula is well-formed, and that every clock used in the formula is defined in AutomatonClocks.

TransitionUpdate (clock updates on transitions): comma separated list of clock assignments according to Definition 2.1.5.[5] Enabled for Transition. Default value "" (i.e., all clocks keep their value). Syntactic check that no clock is assigned more than once, and that every clock used in an assignment is defined in AutomatonClocks.

TCADataConstraints (data guards on transitions): data constraint formula according to Definition 2.1.7. Enabled for Transition. Default value `true`. Syntactic check that every port used in the data constraint is defined in Transi-

[4] See above, every extension is enabled for exactly one of Automaton, State, Transition.

[5] More concretely, write `x`=n for an update $\lambda(x)=n$, and `x`=`y` for an update $\lambda(x)=y$. Clocks not mentioned are assumed to keep their value.

tionPortNames, and every memory cell used in the data constraint is defined in StateMemoryExtension of either source or target location of the transition.

Despite the fact that a data constraint extension for CA already existed (ConstraintExtension), we had to provide the new extension TCADataConstraints for data guards in TCA. The reason is that both types of data constraints provide distinct features, which are not supported by the other data constraint type. For example, CA data constraints allow to reason about functions on the data values, which is not supported in TCA. On the other hand, CA data constraints require that every memory cell used by the target location of the transition is initialised in the data constraint, while for TCA, we do not impose this restriction (cf. Remark 2.3.2).

In the `plugin.xml` file, we defined a number of constraints on (in)admissible and required combinations of extensions, as shown in Table 5.4. We also defined a new automaton type *Timed Constraint Automaton*. When a new automaton of this type is created in the editor, it has the aforementioned extensions (except of course for ConstraintExtension) enabled.

Extension	Requires Extension	Mutually exclusive with
TransitionGuard	AutomatonClocks, TransitionUpdate	
TransitionUpdate	AutomatonClocks, TransitionGuard	
StateInvariant	AutomatonClocks	
TCADataConstraints		ConstraintExtension (CA data constraints)

Table 5.4: Dependencies between extensions in the TCA plugin

The syntactic checks described above are executed in methods *parseExtension* , *editExtension* and *validateExtension* (cf. Figure 5.3). For these checks, the new extensions use the helper classes TCAClocksParser and TCADataParser, contained in package *cwi.ea.extensions.clocks.parsers*. These parsers are generated from the grammar files `TCAClocks.g` and `TCAData.g` using the parser generator ANTLR [ANT].

5.1.3 From TCA to Formulas

We have implemented the translation of TCA to propositional formulas with linear arithmetic, as presented in Section 3.1.3, in ECT. In this section, we describe the implementation of our TCA formula generation.

ECT provides support for code generation (into an arbitrary target language) in package *cwi.codegen*, cf. Figure 5.5. A new code generator is defined by implementing the interface *ICodeGenerator* (or extending one of its abstract subclasses). Interface *IGenModel* encapsulates the data needed for code generation, that means, the properties (content information) of the underlying model that influence the generated code. It offers methods to manipulate these properties. Properties are

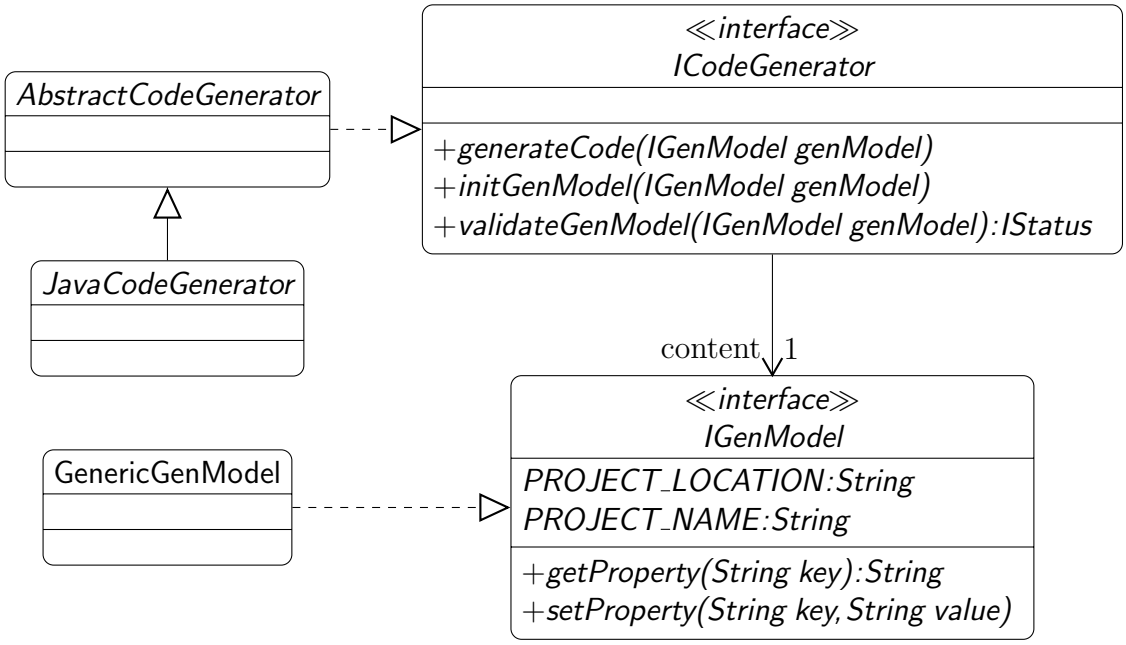

Figure 5.5: Package *cwi.codegen*

key/value pairs of type String, new properties can be easily defined by calling *setProperty(key,value)* on an (until then) undefined key. *IGenModel* defines the two properties *PROJECT_LOCATION* and *PROJECT_Name* which every code generator should support.

The sources for the TCA formula generation extension are located in package *cwi.ea.extensions.clocks.codegen*, cf. Figure 5.6. Interface *SMTFormulaTemplate* can be seen as the main class of the extension. It defines a generic template for intermediate representation of TCA, and serves as a superclass for template classes that generate formulas for different back-ends. Currently, there exists only one class that implements *SMTFormulaTemplate* in package *cwi.ea.extensions.clocks.codegen*, MathSATFormulaTemplate. This class is used to generate input files for the MathSAT [mat] tool.[6]

For each of the constituents φ^X of the formula representation (3.16) (cf. Table 3.4 on Page 50), *SMTFormulaTemplate* defines a method signature *buildX* to generate the corresponding formula: *buildInit()* for φ^{init} (3.10), *buildTrans()* for φ^{trans} (3.13), *buildLocation()* for $\varphi^{location}$ (3.14), and *buildMutex()* for φ^{mutex} (3.15). The boolean parameter *unfold* in the latter three is used to determine whether to generate the step-abstract variant of the formula, that means with indices `t` and `t+1`, or the unfolded variant (cf. Section 3.2.2). In the former case, the return value should be a singleton list, in the latter case, the list should contain one entry for every unfolding depth, sorted in ascending order. The boolean parameter *product* in *buildTrans()* determines whether to generate delay transitions for products of TCA, cf. (3.17). Subclasses of *SMTFormulaTemplate* will have to implement these methods such that they generate formulas in the appropriate input format for the intended back-end solver, while following the constraints explained above.

[6]For MathSAT version 4.2.17.

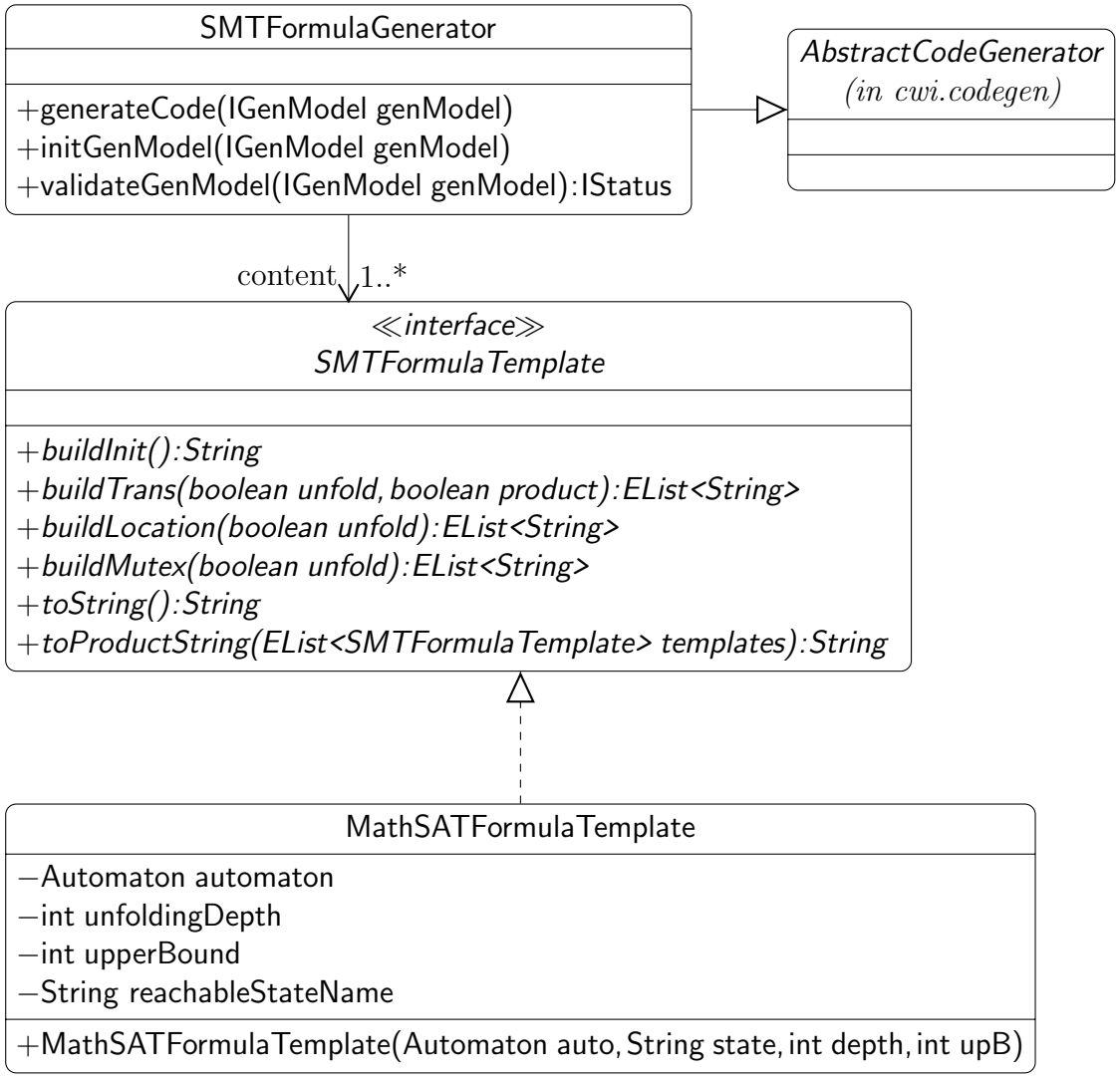

Figure 5.6: Package *cwi.ea.extensions.clocks.codegen*

The formula representation (in String format) for a single *SMTFormulaTemplate* object is obtained by calling method *toString()* . Method *toProductString()* is used to generate the product representation for a list of templates. Subclasses of *SMTFormulaTemplate* will typically implement these methods such that they call the *buildX* methods (passing the runtime value of *unfold* to the latter three), add additional information for the intended back-end solver as needed, and possibly sort the formulas in a specific way (cf. Section 4.2). Any implementation of the *toProductString()* method should be robust enough to produce the same result, independently of whether the object on which it is called (`this`) is contained in the list or not.

The constructor of MathSATFormulaTemplate is used to initialise the private fields with the appropriate values. It can be used for both finite and infinite data domains: for finite data domains, a lower bound of 0 is assumed by default. If the fourth parameter, upB ("upper bound"), is 0 as well, the data domain is assumed to be infinite. Otherwise, upB is required to have a value $\geq$1, which then determines the upper bound of the data domain (cf. also Remark 3.1.6 on finite data domains).

Parameter state can be used to specify the name of a location to be checked for k-step reachability, cf. Section 3.2.3; it is stored in field reachableStateName. The implementation of toString() respectively toProductString() generates the corresponding formulas. If parameter state is the empty string, no such property is generated.

Invocation of the formula generation from the editor (cf. Section 5.2.2) creates an instance of class SMTFormulaGenerator, which is a subclass of *AbstractCodeGenerator* from *cwi.codegen*, cf. Figure 5.6. SMTFormulaGenerator implements the methods from *ICodeGenerator* (cf. Figure 5.5) as follows. Method initGenModel() initialises the properties of the genModel required for formula generation: unfolding depth, data domain, location of the resulting output file, the set of automata to translate to formulas (this is a subset of all automata contained in the currently open file), and the target language (currently, only MathSAT format is supported). These settings are determined from user input to the formula generation wizard pages, cf. Figures 5.13, 5.14 and 5.15.

Method validateGenModel() checks that the genModel initialised in this way satisfies a number of additional (with respect to the constraints described in Section 5.1.2) syntactic and semantic requirements. This is needed for the resulting formulas to be well-defined: formula generation can be invoked for any EA automaton, but the results are well-defined only for TCA. The constraints include for example uniqueness of names, appropriate choice of finite data domain with respect to data values used in data constraints, or appropriate choice of enabled extensions (according to Section 5.1.2).

Method generateCode() starts the actual formula generation. It creates an *SMTFormulaTemplate* instance for each TCA selected for formula generation. The target language property determines the runtime type of these objects, that means which subclass of *SMTFormulaTemplate* to use. Actual parameters of the constructor of the appropriate runtime class are determined from the properties of the genModel. generateCode() then calls either toString() or toProductString(), depending on whether one or more TCA were selected for formula generation, and writes the resulting string to a file, using the corresponding property of genModel to determine the location.

5.1.4 Abstraction Refinement

Abstraction and refinement of TCA is not directly part of the ECT, in that it is not implemented as a plugin or extension within the graphical ECT interface. Instead, it is implemented as a standalone program, which performs abstraction and refinement on a file obtained from the ECT plugin (as explained in the preceding Section 5.1.3), and calls the preferred (corresponding) SMT solver for the verification part. The reason is that many SMT solvers already offer a graphical user interface (MathSAT does not, though). If needed, abstraction and refinement can be integrated into ECT on a command line basis, similar to the integration of mCRL2 (cf. [Kra11]), but we consider this less useful. The sources are found in package *cwi.ea.extensions.clocks.absref*, cf. Figure 5.7.

The interface *TCAAbstractor* serves as a base class for all implementations of abstraction functions working on TCA, independent of a specific target language. It defines method signatures that need to be supported by all such abstractors.

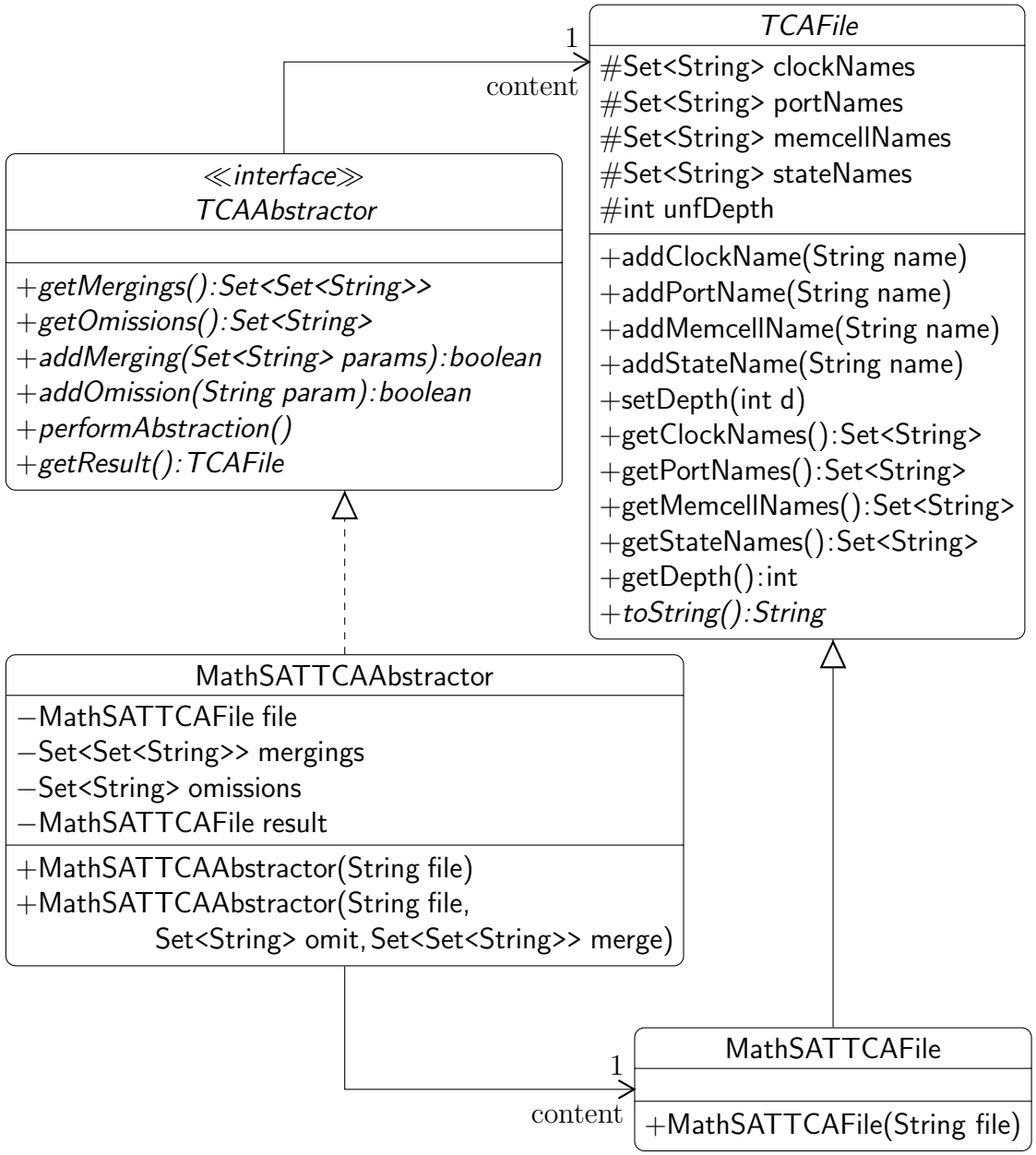

Figure 5.7: Package *cwi.ea.extensions.clocks.absref*

Implementing class MathSATTCAAbstractor is tailored to work on files containing formulas in MathSAT format. In particular, MathSATTCAAbstractor implements the performAbstraction() method, by essentially implementing abstraction function α presented in Figure 4.1 in Section 4.1.

The abstract container class *TCAFile* encapsulates the intermediate representation of TCA in the process of abstraction, independent of a specific target language. It is used for both intermediate and final results.[7] Though *TCAFile* is abstract, it provides implementations for all methods except *toString()* , since this is the only method that depends on the target language, all other operations are performed on the internal/intermediate representation. Class MathSATTCAFile extends *TCAFile*, by implementing toString() such that the resulting String is in proper MathSAT format. Parser class MathSATFormatParser (not shown in Figure 5.7) is used by MathSATTCAAbstractor to parse a text file in MathSAT format and generate a MathSATTCAFile object from it. The text file can be obtained either from ECT, as explained in Section 5.1.3), or be generated by the toString() method of MathSATTCAFile itself. Class MathSATFormatParser is generated from grammar file `MathSATFormat.g` using the parser generator ANTLR [ANT].

Finally, class TCAMain uses the aforementioned classes to provide a little (command-line based) application to perform interactive abstraction refinement. Essentially, the application implements the three steps of the abstraction refinement paradigm (cf. the beginning of Chapter 4), looping through steps two and three. Figure 5.8 shows a conceptual overview of the application, where grey boxes indicate calls to external tools. The implementation is completely interactive, that means the user has to choose the initial abstraction (the application shows the list of abstractable parameters, though) as well as a refinement option and parameter to be refined (the application shows a list of potentially responsible parameters as well as the current counterexample, though). The type of the input file, and thus the SMT solver to be called, are determined from the input file ending. At the moment, only MathSAT is supported (file ending `.msat` or `.mathsat`), but it is easy to extend TCAMain to support other file types and solvers.

Notice that the abstraction refinement loop in Figure 5.8 has two exit points: either the conjunction of property and abstract system is unsatisfiable, in this case, the property holds for k steps (cf. Section 3.2.3); or the conjunction of original system, property and witness run is satisfiable, in this case, a counterexample to the property has been found.

5.2 Workflow

In this Section, we describe the typical workflow when working with the TCA plugin, using the FIFO buffers with expiration from Examples 2.3.3 and 2.3.10. The Section

[7] It would have been possible to extend interface *SMTFormulaTemplate* from package *cwi.ea.extensions.clocks.codegen* (cf. Figure 5.6) in such a way that it can be used for intermediate results of abstraction as well. Yet, this would have prevented us from providing method implementations in abstract class *TCAFile*. Moreover, we prefer to keep the approach modular and flexible, by maintaining two intermediate formats (*SMTFormulaTemplate* for code generation, and *TCAFile* for abstraction).

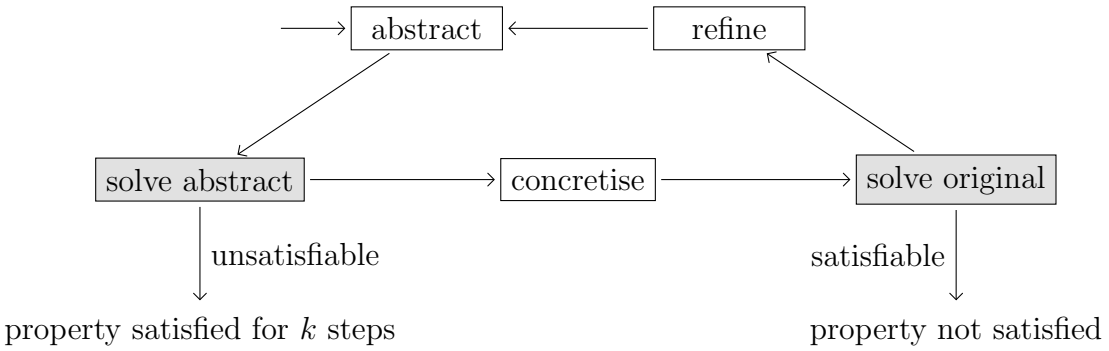

Figure 5.8: Implementation of Abstraction Refinement Loop, Conceptual Overview

can also be seen as a little "Getting started" tutorial to the TCA plugin. We assume that ECT and in particular the EA framework is installed already.[8] For instructions how to install the plugins, please refer to the ECT website `http://reo.project.cwi.nl`.

5.2.1 Editing

The first step is to draw the TCA. From the Eclipse workbench, create a new *General Project* (*File* → *New* → *Project* → *General* → *Project*) with name `Workflow`. In the project, create a new EA automaton file (*File* → *New* → *Other*, scroll down to the *Reo* wizard and choose *Automaton*), and name it `Workflow.ea`. From the palette that appears on the right, choose *Automaton* and left-click on the empty canvas to create a new EA automaton. A menu appears that allows to choose one of the predefined automaton types. If none of the types is chosen, the new automaton has no extensions enabled. Choose *Timed Constraint Automaton* and give it the name `Buffer`. A rectangle, indicating the drawing area for the new automaton, appears on the canvas, cf. Figure 5.9.

The upper part of the drawing area contains general information: the name of the automaton, and information from extensions which are applicable on automaton level. The two symbols below the name indicate that extensions AutomatonPortNames and AutomatonClocks are already enabled for this TCA (these extensions were enabled automatically when automaton type *Timed Constraint Automaton* was chosen). The StateMemoryExtension is not enabled by default. To enable it, right-click on the automaton, choose *Extensions* from the context menu, and check *State Memory*. Since StateMemoryExtension is not applicable to Automaton, there is no visible change. Add two ports `p` and `q` and a clock `x` to the automaton: click on the respectively label until a text field shows up, then enter the text `p,q` respectively `x`.

Next, we add locations. From the palette on the right, choose *State*, and click inside the (lower part of the) rectangle to add a location. Give it the name `empty`. The ingoing arrow indicates that StartStateExtension is enabled for this automaton

[8] All descriptions and images in this section are based on version 3.2.0 of ECT, and version 3.2.9 of the EA framework.

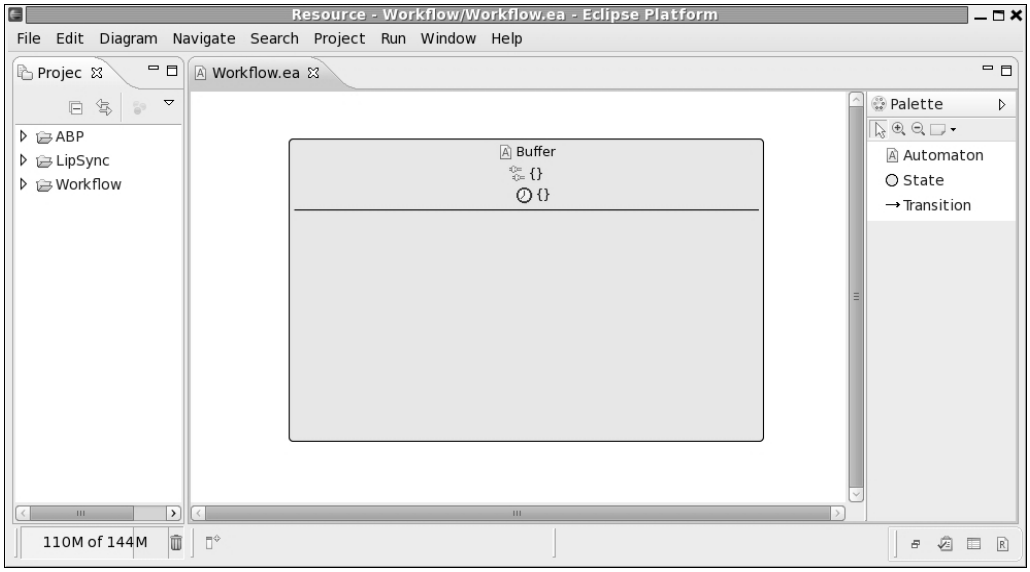

Figure 5.9: Workbench Overview

(again, this extension was enabled automatically when automaton type *Timed Constraint Automaton* was chosen), by default, the first location that is created is set to be the initial location. The initial location can be changed from the context menu of locations. Add a second location with the name `full`. The symbols next to the locations show that StateMemoryExtension ( ) and StateInvariant ( ) are enabled. Add a memory cell `m` to `full`, and set the invariant of `full` to `x<=3`.

To add a transition from `empty` to `full`, choose *Transition* from the palette, click on `empty`, and drag the other end of the transition to `full`. For every (enabled) extension which is applicable to Transition, the new transition has a corresponding label. The four labels are for TransitionGuard, for TransitionUpdate, for TCADataConstraint, and for TransitionPortNames. Set the labels pursuant to Figure 2.5, add two transitions from `full` to `empty`, and set the labels accordingly. Note that memory cell references (`s.m` and `t.m`) in the data guards have to be prefixed with an additional `$` in the editor. This is due to liberal naming conventions in ECT which allow dots `.` to appear in names. The final TCA should look similar to the one shown in Figure 5.10 (in order not to clutter up the picture, we have removed empty transition labels).

Repeat the above steps, and create a second instance of the buffer in the same file `Workflow.ea`. Name it `Buffer2`, with locations `empty2` and `full2`, memory cell `m2` (in `full2`), clock `x2` and ports `q` and `r` (cf. Figure 2.6). Make sure to use `q` on the transition from `empty2` to `full2`. The resulting TCA should look similar to the one shown in Figure 5.11.

Throughout the editing process, every modification is checked for syntactic correctness, according to the syntactic requirements explained in Section 5.1.2. If a syntactic error is detected, the label of the extension (for example or) in which the error occurred is replaced by the error marker . A tooltip appears on mouseover which gives information about the error, and in this way helps to resolve it. As an example, while composing the `Buffer` automaton, we could try to use port `p` on a

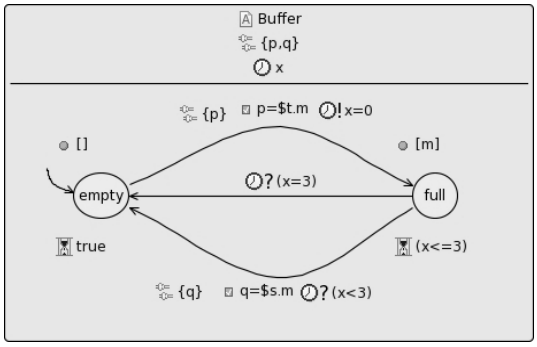

Figure 5.10: First Buffer

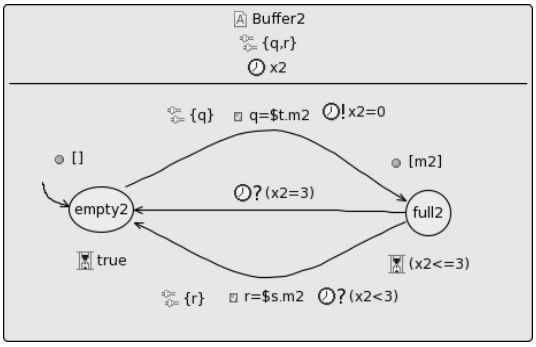

Figure 5.11: Second Buffer

transition *before* declaring it on automaton level. Figure 5.12 shows the resulting error marker and tooltip. To resolve the error, we need to do two things: first, add

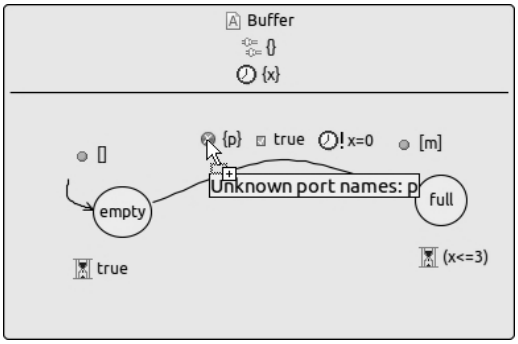

Figure 5.12: Indication of Errors

`p` on automaton level (i.e., in the field with label field just below the name). After that, the error marker remains active, because the editor only checks the extension that has just been edited (AutomatonPortNames in this case). Therefore, we need to edit the TransitionPortNames extension again (simply click on the error marker until the text field appears, and confirm). The TransitionPortNames extension is checked again, and since `p` is now declared on automaton level, the error marker disappears.

5.2.2 Formula Generation

Formula generation is invoked from the context menu of automata: right-click on one of the automata, and choose *Generate code → SMT Formula Generator* to open the code generation wizard.

The first page of the wizard allows to specify the name and directory of the resulting file, the unfolding depth, the range of data values, and the target language (Figure 5.13). Since currently only the MathSAT solver back-end is supported, the only admissible entry in the target language field is `msat`, which naturally is also the default. Enter `Workflow` as *Project name*, `2FIFO.msat` as *Output file name*, and click `Next>` at the bottom of the wizard page.

The second page (Figure 5.14) allows to choose the (combination of) automata to generate code for (the resulting formulas contain delay transitions, cf. Defini-

tion 3.1.8, iff more than one automaton is selected). Select both TCA and click `Next>`.

Finally, the third wizard page (Figure 5.15) allows to select up to one location for each automaton that was selected on the previous (second) wizard page, to check for k-step reachability. Unfold the lists of locations, select locations `full` and `full2`, and click `Finish`. Note that if locations are selected for a (non-empty) subset of automata only, then k-step reachability is checked for all possible combinations of the selected locations with any location from the other automata. For example, if we would select location `full` only, i.e., not select a location for `Buffer2`, reachability would be checked for any of the product locations `(full,empty2)` and `(full,full2)`.

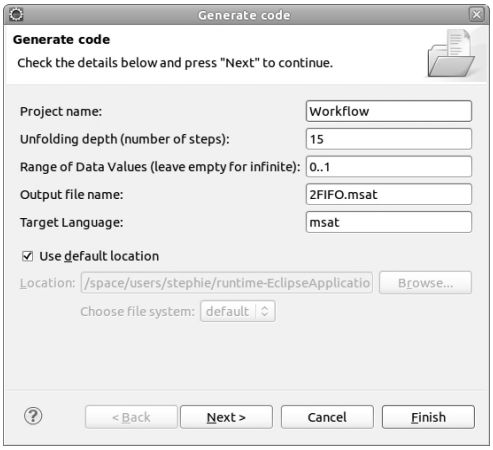

Figure 5.13: Code Generation Wizard, First Page

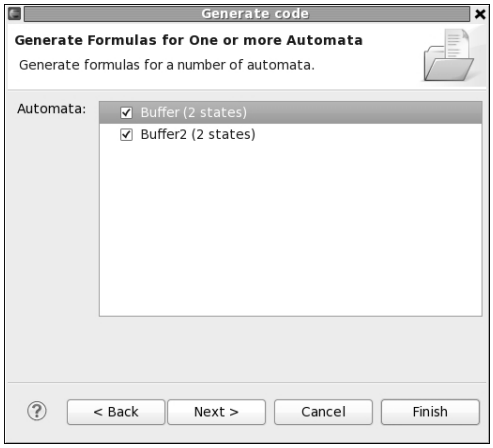

Figure 5.14: Code Generation Wizard, Second Page

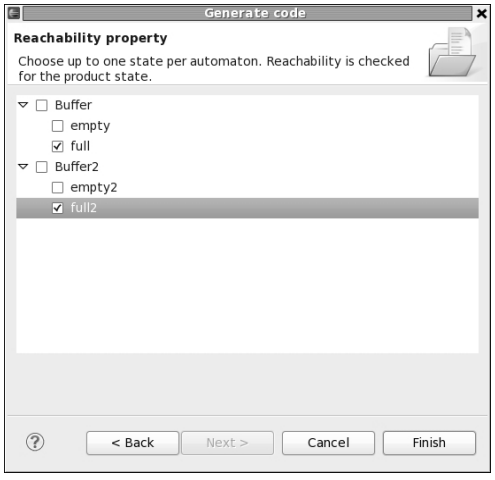

Figure 5.15: Code Generation Wizard, Third Page

After clicking `Finish`, the resulting file `2FIFO.msat` can be found in the *Project Explorer* view to the left of the canvas, it is located in the *src* subdirectory in the

Workflow project, cf. Figure 5.16.

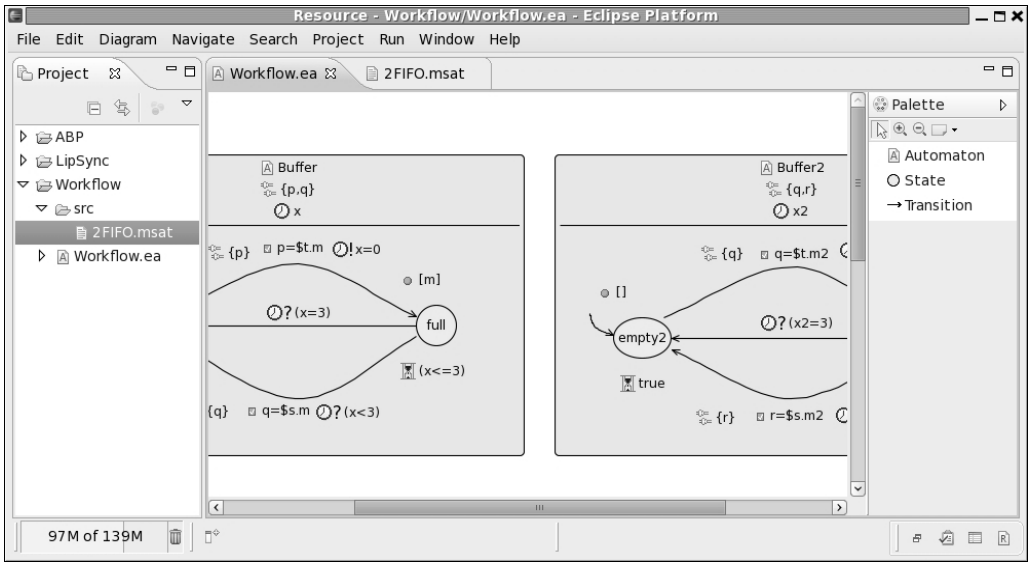

Figure 5.16: Workbench, Project Explorer View

5.2.3 Verification

We now have two options to continue. The first option is to directly call MathSAT on the generated file `2FIFO.msat`. In general, MathSAT is invoked by calling `mathsat [options] input_file` from the command line. In our context, the minimal set of options includes:[9]

`-solve:` tells the solver to solve the formula (other options include for example transformation to CNF)

`-input=msat:` specifies the input format (other options are `smt` or `dimacs`)

`-logic=QF_LRA:` specifies the logic to be used (Quantifier Free Linear Real Arithmetic)

Other useful options include for example

`-print_model:` prints one satisfying valuation (model), if it exists

`-allsat:` prints *all* satisfying valuations, if they exist (only recommended for small problems)

`-outfile=FILE:` redirects the output from stdout to file `FILE`

[9]For a complete and detailed overview and explanation of the available options, please refer to [mat]

The output of a call to MathSAT always starts with some statistical information about the input problem and the involved theory solvers. The essential information can be found at the very end of the output: the last line of the output of the call `mathsat -solve -input=msat -logic=QF_LRA 2FIFO.msat` says `sat`, which means the set of input formulas is satisfiable, and thus the "error state" `(full,full2)` is reachable.

The second option is to call the abstraction refinement application (cf. Section 5.1.4) on the generated file `2FIFO.msat`: `java TCAMain 2FIFO.msat`. As explained in Section 5.1.4, the application first asks the user to create the initial abstraction. For ports, for example, the application outputs

```
Choose ports to abstract/merge (type numbers of ports
to be merged, separated by blanks, type 'X' to end):
1: r
2: p
3: q
```

and for clocks, it outputs

```
Choose clocks to abstract/remove (type numbers,
separated by blanks):
Set of clocks is
1: x
2: x2
```

Type `1` (and confirm) to remove clock `x`. After this initial abstraction is determined, the application internally calls MathSAT on the abstract system. Since the "error state" `(full,full2)` was already reachable in the original system, it is of course reachable in the abstract system as well. The next output of the application is

```
Spurious counterexample found, abstraction needs to be refined.
Choose refinement option (type number):
1: rule out counterexample trace
2: refine a parameter
```

Type `2`, this gives the output

```
Choose parameter to refine (type number):
1: x
```

Now type `1` to refine clock `x`. The application refines the abstraction, and calls MathSAT again on the resulting system (which is the original system already). The next (final) output is

```
Property does not hold.
```

and the application terminates.

5.3 Case Studies

As stated in Chapter 2, TCA are specially tailored to implement coordinating connectors in networks where timed components communicate by exchanging data through multiple channels. In this way, TCA impose a certain communication pattern on associated components, that means, they force the components to behave (communicate) in a certain way. An obvious application area is therefore to use TCA for modelling time- and data-aware communication protocols.

In this section, we describe two such protocols, present experimental results we got from modelling and analysing the protocols with our TCA plugin for ECT, and discuss the benefits of using TCA as underlying formalism for these case studies and in general.

5.3.1 Alternating Bit Protocol

In this section, we present the *alternating bit protocol* (ABP). The ABP is a network protocol, which ensures successful transmission of data elements between a sender and a receiver over unreliable channels. It is one of the standard benchmarks in the context of component based systems and process algebra, and has been discussed in detail for example in [Mil89, Fok00, LM87].

Essentially following the description in [Mil89], we design the protocol from four subcomponents: the Sender, the Receiver, and two unreliable channels Channel1 and Channel2 connecting the former two. Each of these components is modelled with a separate TCA. We assume that the channels may loose, but not corrupt or duplicate, data at random. Yet, it is very easy to change this behaviour, simply by exchanging the TCA that model the channels. For an overview of how to model timed channels with different behaviour, see for example [ABdBR07]. A conceptual overview of the components is shown in Figure 5.17. Arrows indicate the intended direction of dataflow, labels indicate the ports through which the components communicate.

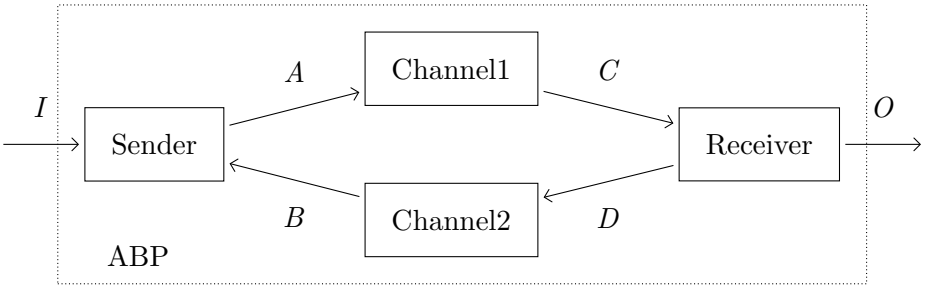

Figure 5.17: ABP Connector, Conceptual Overview

The protocol works as follows: after accepting input (from the environment) through port I, the Sender starts a timer, and sends the message to the Receiver via Channel1, i.e., it sends the message to Channel1 through port A, and Channel1 in turn sends the message to the Receiver via port C. The Sender attaches a control bit b to the message, and expects the Receiver to send back the corresponding control bit b through Channel2 as acknowledgement. After having received the

acknowledgement, the Sender is ready to accept another input from the environment, which it sends to the Receiver with attached control bit $\neg b$ (this is where the name *alternating* stems from). If the timer of the Sender expires before it receives the acknowledgement bit b, or if it receives acknowledgement bit $\neg b$ (which it ignores), it assumes an error has occurred, resets the timer and resends the message with bit b.

The Receiver works complementary: it receives a message, together with a control bit b, from the Sender through Channel1. After delivering the message to the environment through port O, the Receiver sets a timer, and sends bit b as acknowledgement to the Sender through Channel2. Next, it expects a message tagged with bit $\neg b$. If the timer expires, or the next message is tagged with b again (which the Receiver ignores), the Receiver assumes an error has occurred, resets the timer and resends the acknowledgement bit b.

We assume an arbitrary but fixed, finite set of messages $\mathcal{M}sg$. The data domain is $\mathcal{D}ata{=}\mathcal{M}sg{\cup}\mathcal{M}sg{\times}\{0,1\}{\cup}\{0,1\}$. The three subsets of $\mathcal{D}ata$ correspond to messages sent from the environment to the Sender, and from the Receiver to the environment ($\mathcal{M}sg$), messages tagged with control bits sent from the Sender to the Receiver ($\mathcal{M}sg{\times}\{0,1\}$), and acknowledgement bits sent from the Receiver to the Sender ($\{0,1\}$).

The TCA for the Sender, the Receiver and the two channels are shown in Figures 5.18, 5.19 and 5.20. Since ports can only transmit a single data item, we model ports A (between Sender and Channel1) and C (between Channel1 and Receiver) using two ports A_1,A_2 and C_1,C_2, respectively. This is because for a pair $(m,b){\in}(\mathcal{M}sg{\times}\{0,1\})$, we need to be able to reason about the constituents m and b separately. For both Sender and Receiver, we assume a timeout of 2 for resending the message and acknowledgement, respectively. For example, after the Sender has sent a message and has moved to location $wait0$, it waits for acknowledgement bit 0 before moving to location $idle1$. If this bit does not arrive before clock x has reached value 2, the Sender moves back to location $send0$, where it waits at most one more time unit before it resends the message.

The TCA modelling the entire protocol component—we call it $\mathrm{T}_{\mathrm{ABP}}$—is obtained by composing the TCA of the four subcomponents (cf. Definition 2.3.9), that means

$$\mathrm{T}_{\mathrm{ABP}} = (\mathrm{Sender} \bowtie \mathrm{Receiver} \bowtie \mathrm{Channel1} \bowtie \mathrm{Channel2})$$

5.3.1.1 Verification

While the internal behaviour of the ABP ensures reliable transmission of messages over unreliable channels, from the outside, it behaves as a perfect buffer of capacity one (cf. [Mil89]). That is, it accepts and delivers messages from and to the network (through ports I and O, respectively) alternately, and the order of data elements is not changed.

The alternation is described by the LTL formula

$$\Box((I \rightarrow \bigcirc(\neg I \mathcal{U} O)) \wedge (O \rightarrow \bigcirc(\neg O \mathcal{U} I))),$$

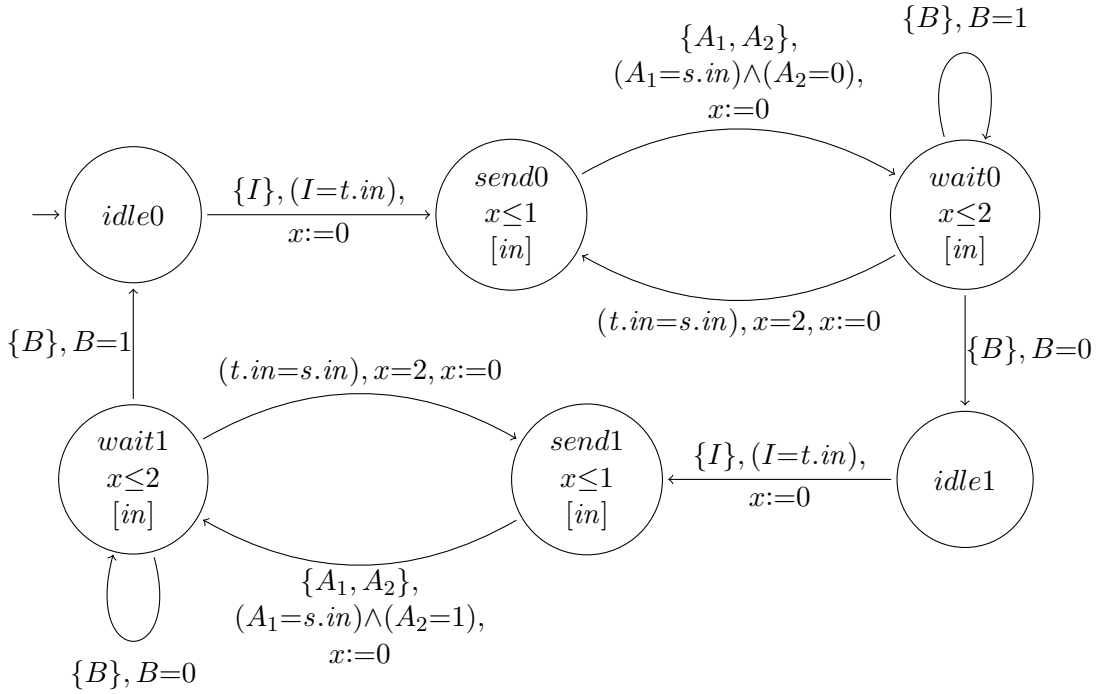

Figure 5.18: ABP, Sender

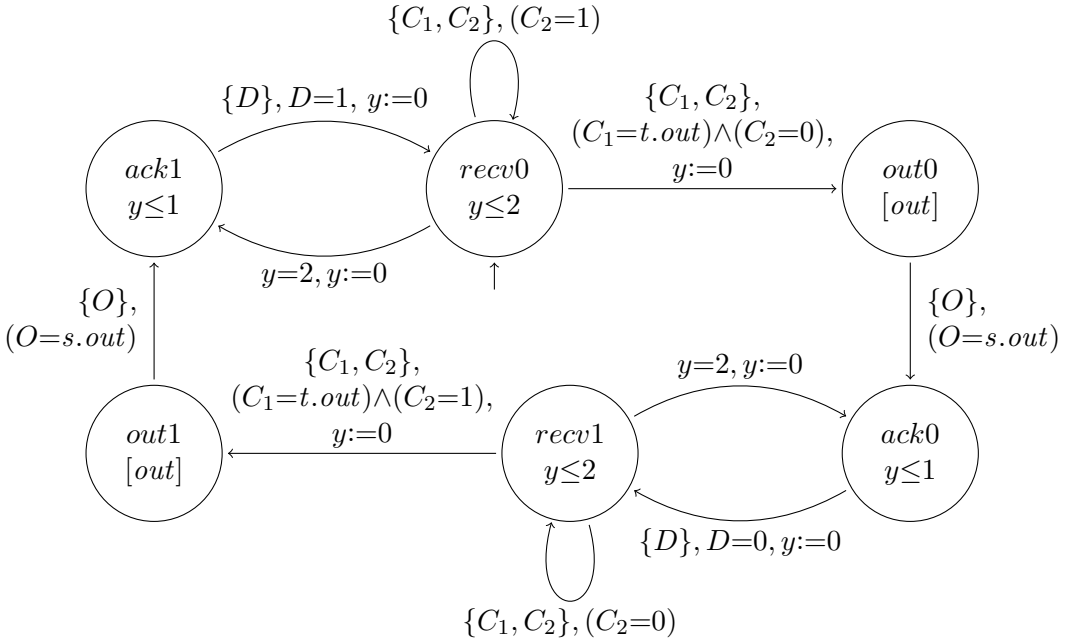

Figure 5.19: ABP, Receiver

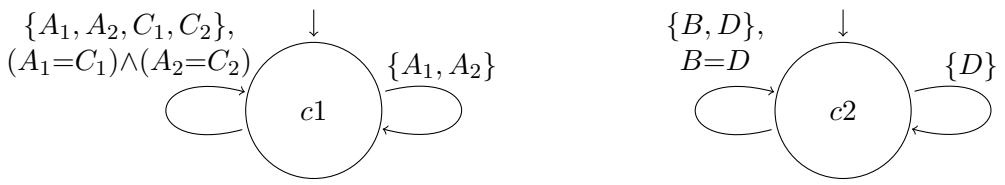

Figure 5.20: ABP, Channel1 (left), Channel2 (right)

which expresses that between any two communications through port I, there is a communication through port O, and vice versa. We call this property `Buffer`.

To check for the correct order of data elements, we identify a set of error locations. First recall that locations in $\mathrm{T}_{\mathrm{ABP}}$ comprise one location for each of the four subcomponents, for example, the initial location of $\mathrm{T}_{\mathrm{ABP}}$ is $(idle0, revc0, c1, c2)$. The error locations are

$$\{(waiti, outi, c1, c2) \mid in{\neq}out, i{=}0, 1\} \cup \{(sendi, outi, c1, c2) \mid in{\neq}out, i{=}0, 1\}$$

Each of these locations corresponds to a configuration where the Sender subcomponent has received a data item through port I and stored it in memory cell $[in]$, but the Receiver subcomponent has stored a different data item in memory cell *out* which it is about to send to the environment through port O. Note that the formulation of this property relies on the first property of alternating dataflow through I and O. We call the property expressing that none of the error states is reachable `Error`.

We have modelled the TCA of the ABP with the ECT plugin. For performance comparison, and to show that our approach scales very well on reachability properties, we compare three unfolding depths $k{\in}\{20, 50, 100\}$. We have generated the corresponding formula representation from within the editor, as described in Section 5.2.2. To show the performance improvements gained from abstraction, we have identified a tailored abstraction function for each of the properties. First observe that timing information can be considered irrelevant for both properties. Moreover, observe that `Buffer` does not rely on exact data values, but only reasons about activity on ports. For `Error`, we define an abstraction function α_1 that removes all timing information from the system: $\mathcal{O}_1{=}\{x, y\}$, and $\gamma_1{=}id$. For `Buffer`, we define an even coarser abstraction function α_2, which in addition removes all information about the exact data values: $\mathcal{O}_2{=}\{x, y, (I{=}t.in), (A_1{=}s.in), (t.in{=}s.in), (C_1{=}t.out), (O{=}s.out)\}$, and $\gamma_2{=}id$.

Table 5.21 shows our experimental results for the different unfolding depths, properties and abstractions. Note that since `Buffer` describes correct alternation of data flow through ports I and O, we need to check the negation of `Buffer`. Abstraction function α_2 removes all information about concrete data values, therefore, the error states are trivially reachable under this abstraction, and we cannot expect `Error` to hold. We have marked the corresponding entries with a slanted font and the additional entry (S) (for "satisfiable"). All other properties are satisfied, which means that the result of verification is "unsatisfiable".

The results in Table 5.21 clearly show that our approach is tailored to reachability properties, and that on these, it scales very well for large unfolding depths. While

	k=20		k=50		k=100	
	¬`Buffer`	`Error`	¬`Buffer`	`Error`	¬`Buffer`	`Error`
$\varphi(\mathrm{T_{ABP}})_k$	1.050s 20.79MB	1.013s 18.85MB	52.50s 61.59MB	13.12s 39.19MB	1690s 508.53MB	80.98s 111.62MB
$\alpha_1(\varphi(\mathrm{T_{ABP}}))_k$	1.704s 20.93MB	0.799s 19.44MB	519.9s 215.02MB	7.082s 32.76MB	segm. fault	32.92s 57.09MB
$\alpha_2(\varphi(\mathrm{T_{ABP}}))_k$	1.430s 19.23MB	*0.175s (S)* *15.99MB*	458.7s 232.60MB	*0.880s (S)* *21.215MB*	segm. fault	*3.984s (S)* *29.11MB*

All experiments have been carried out with MathSAT, version 4.2.17, on an Intel Core2 Duo CPU E4500, with 2.20GHz and 2.5GB RAM

Table 5.21: Experimental Results for the ABP

for k=20, the two properties take around the same time and memory consumption, checking `Error` is factor 4 faster, with factor 1.5 less memory, than `Buffer` for k=50, and almost factor 20 faster, with factor 4 less memory, for k=100. Comparing the same property on different unfolding depths, time and memory consumption increase by factors 50 and 3 (k=20 to k=50), and factors 30 and 8 (k=50 to k=100) for `Buffer`, while these factors are limited to 13 and 2 (k=20 to k=50) and 6 and 3 (k=50 to k=100) for `Error`.

As a second result, Table 5.21 shows the improved performance for `Error` on the abstract system, resulting in a speed-up of factor 1.2 for k=20, factor 1.8 for k=50, and almost factor 2.5 for k=100. Memory consumption is almost the same for k=20 and k=50, but reduces to half for k=100. Note that verification is very fast in case the reachability property `Error` does not hold (i.e., where the input is satisfiable). In contrast to this, performance of `Buffer` on the abstract system decreases, even leading to a segmentation fault for k=100. Though this might seem surprising at first glance, the reason is obvious: since `Buffer` reasons about all possible runs, and the abstract system permits more runs than the concrete system, checking `Buffer` on the abstract system is more expensive. What can be seen though is a slightly improved performance when comparing the two abstractions.

5.3.2 Lip-Synchronisation Protocol

In this section, we describe a *Lip-synchronisation protocol* (LSP). The LSP was first described in the synchronous language Esterel [SHH92]. Later, specifications were presented amongst others using timed LOTOS [Reg93], LOTOS/QTL [BBBC94, BBBC97], timed CSP [ABSS96], timed automata [BFK+98, KLP10], and the Duration Calculus [MLWZ01].

The problem of lip-synchronisation is the following: a presentation device (for example a media player) receives input from two media sources: a Sound Stream and a Video Stream, and should display the two streams synchronously. Yet, small

delays are not human-perceivable, therefore, synchrony needs not be perfect, but certain deviations of the actual presentation times from the optimal presentation times are acceptable. The LSP is used to ensure this degree of synchrony. For this, it has to take into account three major points:

Firstly, the frequencies of the two streams may be different, i.e., there is no (at least not necessarily) one-to-one correspondence between frames of the two streams.[10]

Secondly, the streams may experience a phenomenon called *jitter*. In an ideal scenario, frames are received from the streams with a constant frequency. Yet, the actual arrival times of frames may deviate from these optimal arrival times, for example due to transmission delays. These deviations from the optimal presentation time on *the same* stream are called jitter. Intuitively, the user perceives jitter in case the sound (equivalently video) presentation does not run "smoothly", for example because some parts are skipped or presented too fast.

From [BFK+98], we adopt the notions of *anchored* and *non-anchored* jitter: anchored jitter describes deviations that only depend on the current frame. That means, each frame arrives within a certain interval around *its own* optimal arrival time. Non-anchored jitter, on the other hand, describes deviations that depend on the previous frame. That means, each frame arrives within a certain interval after the *previous* frame.

The last point the LSP has to consider is that the two streams can "drift apart": each frame has an optimal position on the respective other stream. The term *skew* describes deviations from this optimal position on *the other* stream. Intuitively, the user perceives skew in case the sound and video presentation "do not agree", that means if a sound (for example the sound of breaking pottery) is audible significantly before or after the corresponding action (for example a falling mug) is visible.

Summing up, the LSP has to ensure that the two streams are interleaved in the right way, and that the presentation of frames does not violate the acceptable bounds on jitter and skew.

Notation 5.3.1 (Arrival Times of Frames). In the sequel, we use t_i^{opt} to refer to the optimal arrival time of the i-th frame (sound or video), and t_i^{act} to refer to the actual arrival time of the i-th frame.

In line with previous specifications ([SHH92, Reg93, BFK+98]), we design the protocol as follows. A sound frame should be displayed every 30 milliseconds (ms), no jitter is allowed on the Sound Stream, i.e., $t_i^{act}=t_i^{opt}$, and $t_i^{act}=(t_{i-1}^{act}+30)$ for all frames. A video frame should ideally be displayed every 40ms, but we allow non-anchored jitter of ±5ms: $(t_i^{act}-5)\leq(t_{i-1}^{act}+40)\leq(t_i^{act}+5)$. That means, to ensure that a human perceives the media presentation as synchronous, it is sufficient to display each video frame within the interval [35ms,45ms] after the previous frame. The video presentation may precede the sound presentation by 15ms (skew), and may lag behind by 150ms. The fact that jitter is not allowed on the Sound Stream actually allows us to

[10] As pointed out in [BFK+98], if there was a one-to-one correspondence, the obvious solution to achieve synchrony would be to multiplex the streams for transmission, and demultiplex them at the presentation device.

interpret skew as anchored jitter on the Video Stream: $(t_i^{act}-15) \leq t_i^{opt} \leq (t_i^{act}+150)$. We restrict the models of the streams to the arrival times of frames received from the streams. Similarly, we do not model the presentation device, but assume that frames are presented when the corresponding signal (from the LSP) occurs. Further, we assume that the presentation of a video frame can be delayed for an arbitrary amount of time if the frame arrives too early.

We model the protocol with five TCA: Sound Manager, Video Manager, Skew Observer, Jitter Observer, and Initialiser. The design of the TCA is inspired by the timed automata in [BFK+98]. Yet, due to the extended modelling power of TCA, we obtain a more concise model (cf. also Section 5.3.3), in particular for the Skew Observer, which we model using a single counter (rather than using multiple clocks in [BFK+98]). A conceptual overview of the protocol components is shown in Figure 5.22. We omit the Initialiser, since its only purpose is to start the protocol components in the right order. The presentation device is added to Figure 5.22 for illustration only, it is not part of the protocol. As for the ABP (Figure 5.17), arrows indicate the intended direction of communication, labels indicate the ports through which the TCA communicate.

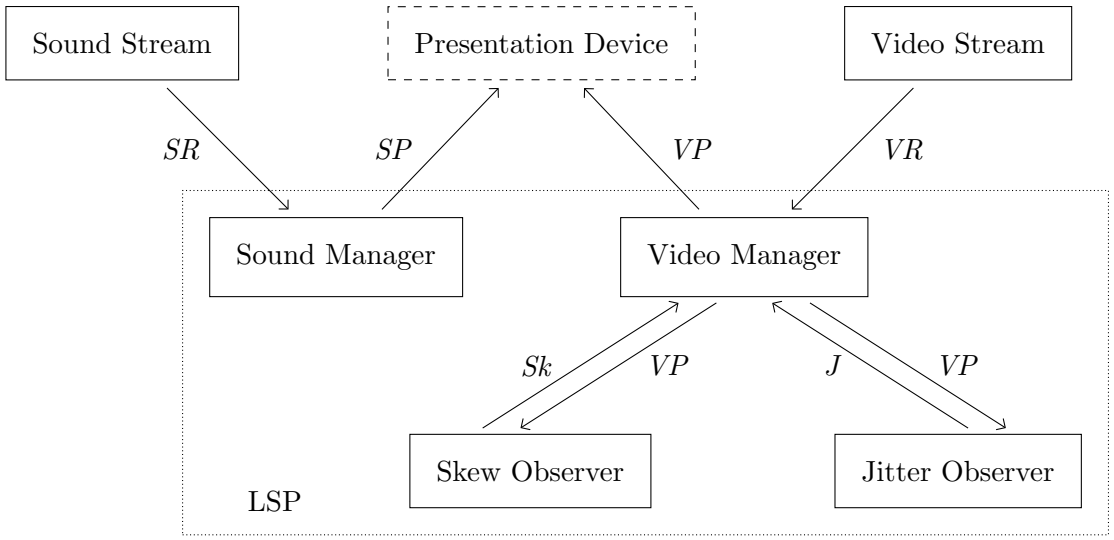

Figure 5.22: LSP: Conceptual Overview

Since the TCA of the LSP (and their interaction) are more complex than the TCA of the ABP, we now describe each TCA in detail. In the end, the TCA modelling the protocol component—we call that TCA T_{LSP}—is obtained by composing the TCA of the five subcomponents (cf. Definition 2.3.9), that means

$$T_{LSP} = (\text{Initialiser} \bowtie \text{SoundManager} \bowtie \text{VideoManager} \tag{5.1}$$
$$\bowtie \text{SkewObserver} \bowtie \text{JitterObserver}) \tag{5.2}$$

5.3.2.1 Sound Manager

The conditions on presentation of sound are very strict: a sound frame should be presented every 30ms, and no jitter is allowed on the Sound Stream. The task of

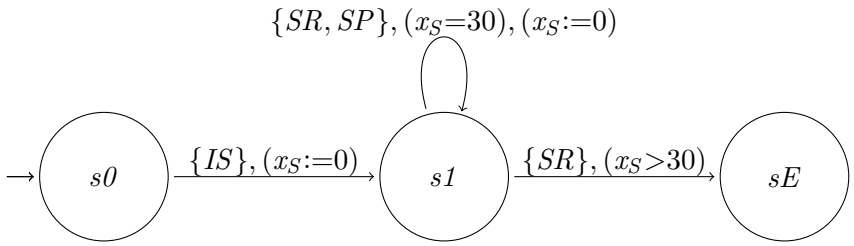

Figure 5.23: LSP, Sound Manager

the Sound Manager (Figure 5.23) is to ensure this behaviour. It uses a clock x_S to measure the time distance between two subsequent sound frames. After being initialised by the Initialiser (cf. Section 5.3.2.5) through port *IS* on presentation of the first sound frame, the Sound Manager starts its cyclic behaviour, waiting for a sound frame to be ready (signalled through port *SR*) every 30ms, and sending to the presentation device the order to present the sound frame (through port *SP*) at the same moment. If the signal that a sound frame is ready arrives late, the Sound Manager moves to its error location *sE*.

For explanatory purposes, in the sequel we use the terms "presentation of a sound frame" and "communication through port *SP*" interchangeably.

5.3.2.2 Video Manager

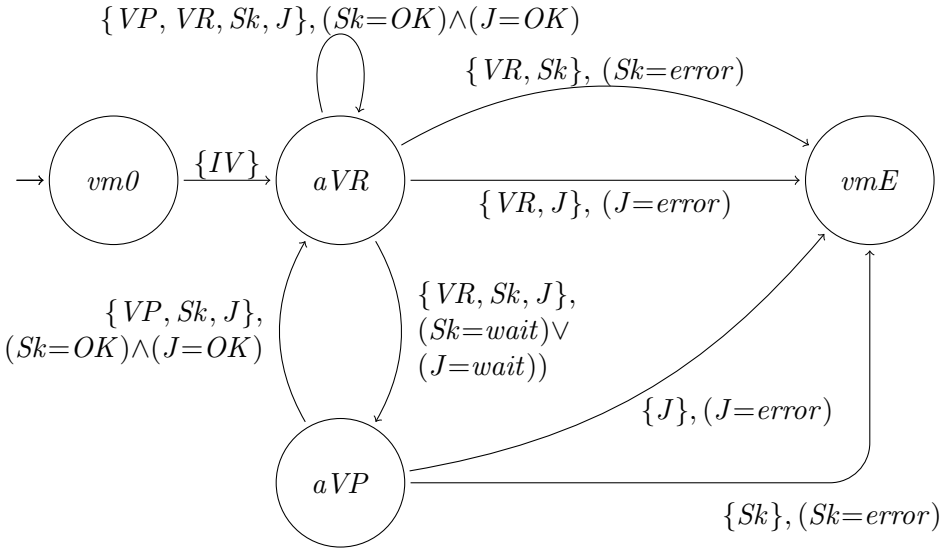

Figure 5.24: LSP, Video Manager

The Video Manager (Figure 5.24) ensures that the timing of video presentation conforms to the bounds described above. Yet, the Video Manager does not control the timing itself, but for this consults the two "helper" components Jitter Observer (cf. Section 5.3.2.3) and Skew Observer (cf. Section 5.3.2.4). In particular, the Video Manager does not use any clocks. The Video Manager is initialised by the Initialiser (cf. Section 5.3.2.5) through port *IV* when the first video frame is presented. In

location *aVR*, it awaits a signal *VR* from the Video Stream, indicating that the next video frame is ready. When receiving this signal, the Video Manager checks the timing conditions with the Skew Observer through port *Sk*, and with the Jitter Observer through port *J*. If both return *OK* (loop in *aVR*), the next video frame can be presented immediately, which is signalled to the presentation device through port *VP*. If either of the observers returns *wait*, video presentation is too early and needs to be delayed. In this case, the Video Manager moves to location *aVP*, and waits until both observers return *OK*. If at any point, either of the observers returns *error*, the Video Manager moves to its error location *vmE*.

For explanatory purposes, in the sequel we use the terms "presentation of a video frame" and "communication through port *VP*" interchangeably.

5.3.2.3 Jitter Observer

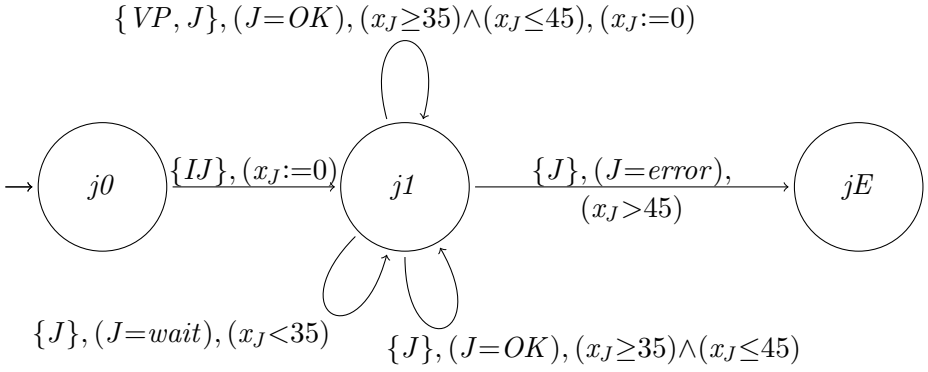

Figure 5.25: LSP, Jitter Observer

The Jitter Observer (Figure 5.25) checks that non-anchored jitter on video presentation remains within the acceptable bounds, that means, that every video frame is presented within an interval of [35ms,45ms] after the previous frame. It uses clock x_J to measure the time distance between two subsequent frames. When the Jitter Observer is initialised by the Initialiser (cf. Section 5.3.2.5) through port *IJ* on presentation of the first video frame, it resets its clock to zero, and moves to location *j1*. The Video Manager can now request the current status of jitter—i.e, whether presenting a video frame at the current point in time would conform to the bounds—by communicating with the Jitter Observer through port *J*. Depending on the value of x_J, the Jitter Observer returns either *wait* (lower left loop in *j1*), *OK* (lower right loop and upper loop in *j1*) or *error* (transition from *j1* to *jE*). If the Video Manager communicates through both ports *J* and *VP*, it request the status of jitter and simultaneously sends the order to present a video frame, which is only possible if presentation is acceptable. In this case, the Jitter Observer resets its clock x_J to start the timer for the next frame.

Note that by construction, the Jitter Observer can only enter its error location *jE* if the Video Manager enters its error location *vmE* at the same time.

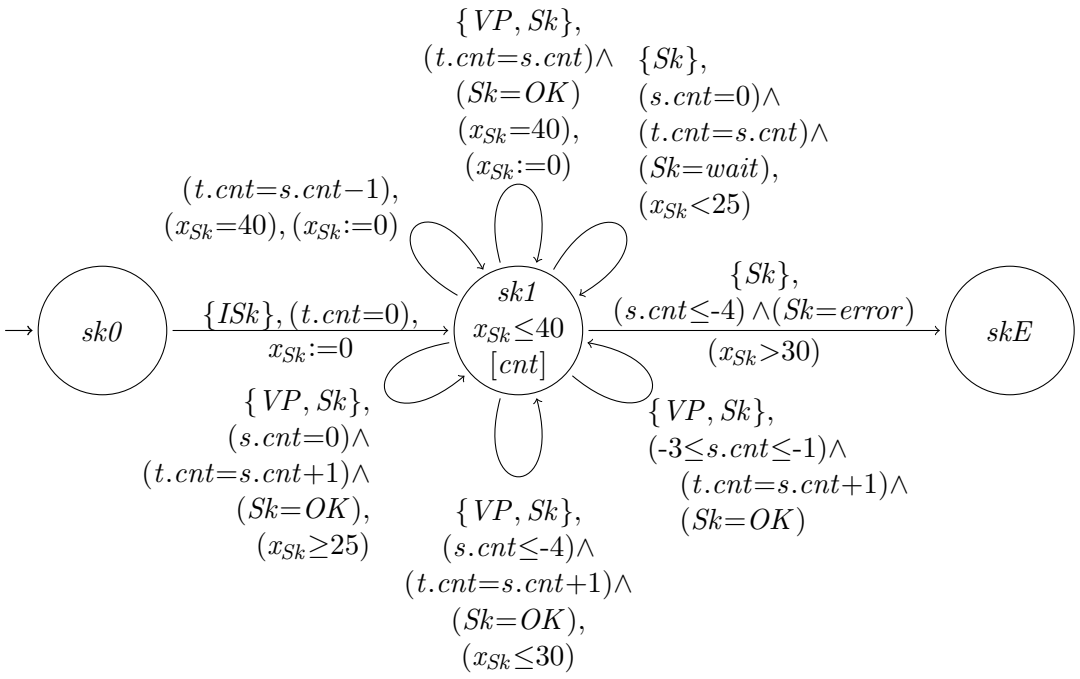

Figure 5.26: LSP, Skew Observer

5.3.2.4 Skew Observer

The task of the Skew Observer (Figure 5.26) is to measure the skew (i.e., non-anchored jitter, see above) on video presentation. For this, the Skew Observer uses a clock x_{Sk} and a memory cell *cnt*. Clock x_{Sk} is used to determine the optimal presentation time t_i^{opt} for video frames. The counter *cnt* is used to calculate—together with x_{Sk}—the exact amount of skew at any given point in time. The Skew Observer is initialised by the Initialiser (cf. Section 5.3.2.5) through port *ISk* on presentation of the first sound frame. When this communication occurs, the Skew Observer resets x_{Sk} to zero, sets *cnt* to zero, and moves to location *sk1*.

The general idea of *cnt* can be roughly described as keeping track of the "overflow" of video frame presentations in case the presentation lags behind by more than one period (of 40ms). In detail, this works as follows: every 40ms (measured by x_{Sk}), that means every time a video frame *should* be presented, the value of *cnt* is decreased by one (upper left loop in location *sk1*), and every time a video frame *is* presented, it is increased by one (lower three loops in location *sk1*). In this way, a negative value of *cnt* indicates that $t_i^{act} > t_i^{opt}$ for the last (i-th) video frame, i.e., video lags behind its ideal position; equivalently, a positive value of *cnt* indicates that $t_i^{act} < t_i^{opt}$ for the last (i-th) video frame, i.e., video is ahead of its ideal position. The exact values of clock x_{Sk} and memory cell *cnt* are used to determine *how much* the presentation is ahead of/behind its ideal position. To explain how this works, we look at the following three situations: on presentation of a video frame, *cnt* is (1) incremented to 1, (2) incremented to zero, and (3) incremented to a value <0.

When *cnt* is incremented to 1 on presentation of the i-th video frame at time t_i^{act}, video is ahead of its ideal position (since $cnt>0$). This ideal position is at time

t_i^{opt}, which is the next point in time when x_{Sk} reaches 40.[11] The amount of time by which video is ahead of its ideal position at time t_i^{act} is thus given by the difference between the current value of x_{Sk} and 40. Therefore, if a video frame is about to be presented when cnt=0 (i.e., cnt would be set to 1) and x_{Sk}<25, the presentation of the video frame would be more than 15ms (40−25) early, and thus needs to be delayed (upper right loop in location *sk1*). As soon as $x_{Sk}\geq$25, the presentation time is within the acceptable bounds, and the video frame can be presented (lower left loop in location *sk1*). An example for this is depicted in Figure 5.27, showing a situation where each video frame is presented as early as possible: if the fourth video frame would be presented after 140ms, it would be 20ms too early, so the presentation needs to be delayed for at least 5ms.

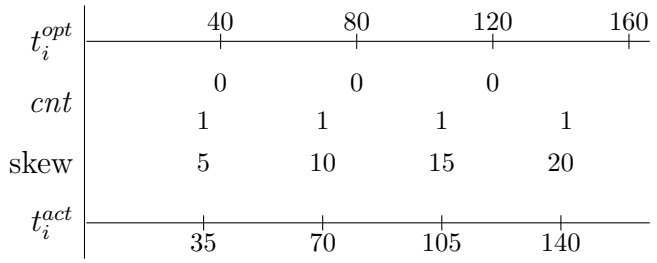

Figure 5.27: Video Presentation ahead: Example

When cnt is incremented to zero on presentation of the i-th video frame at time t_i^{act}, video lags behind its ideal position (since cnt was -1 just before, see above), and the amount by which it lags behind is given by the difference ($t_i^{act}-t_i^{opt}$), which is equal to the value of x_{Sk} at time t_i^{act}.[12]

Finally, when cnt is incremented to a value <0 on presentation of the i-th video frame at time t_i^{act}, video lags behind its ideal position by more than one period (i.e, more than 40ms). Figure 5.28 shows an illustration of this: if each video frame is presented as late as possible, after 405ms, cnt is incremented to -1, and video is 45ms (more than one period) late by that time. In general, if cnt is incremented to -k, video lags behind its ideal position by $k*40+x_{Sk}$. As an example, consider Figure 5.28 again: at time 405, the last time x_{Sk} was reset was at time 400, that means 5ms before, and video presentation lags behind by $k*40 + x_{Sk}=45ms$. The lower loop in location *sk1* thus represents the last possible time where it is still acceptable to present a video frame: on execution of the transition, cnt is incremented to -3, that means video lags behind by -3∗40 + x_{Sk}, and since the clock guard only permits values $x_{Sk}\leq$30, video lags behind by at most 150ms. In case cnt would be set to -3, and x_{Sk}>30 on presentation of a video frame, an out-of-synchronisation error has occurred (transition from location *sk1* to *skE*).

The remaining loops in location *sk1* represent the case where video lags behind, but within the acceptable bounds (lower right loop), and the case where a video

[11]Note that this is indeed t_i^{opt} (and not t_j^{opt}, with $j< i$), since otherwise cnt would have been set to a value >1.

[12]Note that the last time x_{Sk} was reset was indeed t_i^{opt} (and not t_j^{opt}, with $j>i$), since otherwise cnt would have been set to a value <0.

frame is presented at an optimal presentation time (upper middle loop), that means t_i^{act}=t_j^{opt}. Note that apart from i=j, i<j is possible as well in case video presentation lags behind, but i>j is not possible due to the bound of 15ms acceptable for video presentation being ahead (cf. Figure 5.27 again).

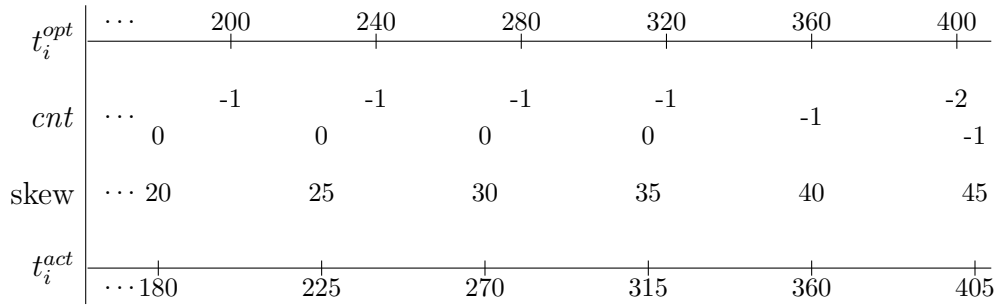

Figure 5.28: Video Presentation behind: Example

Note that by construction, the Skew Observer can only enter its error location *skE* if the Video Manager enters its error location *vmE* at the same time.

5.3.2.5 Initialiser

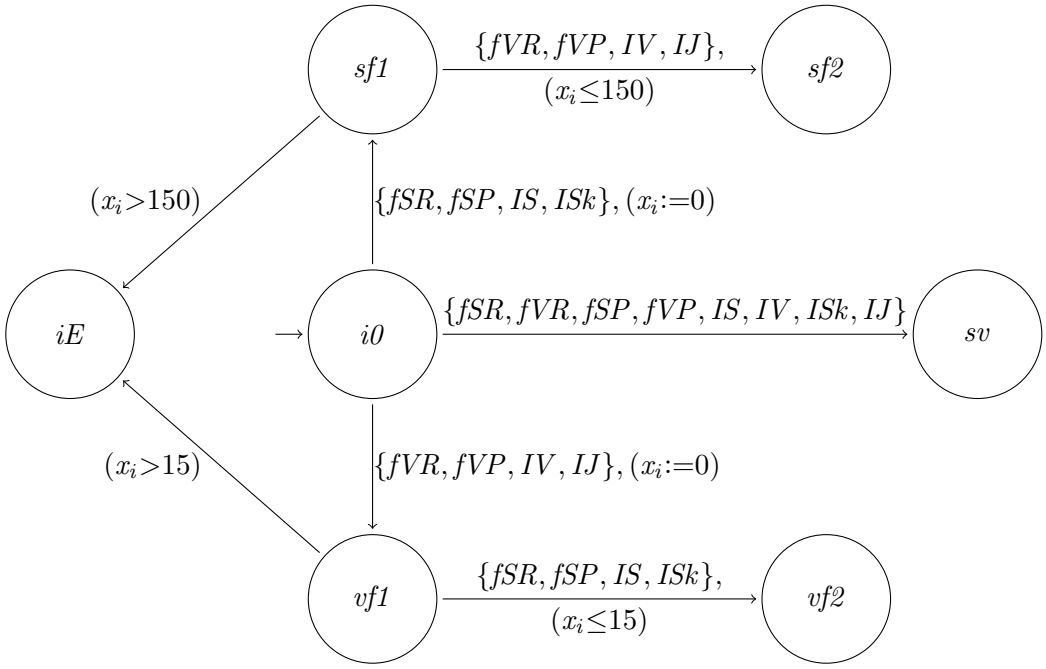

Figure 5.29: LSP, Initialiser

The task of the Initialiser (Figure 5.29) is to start the other protocol components in the right order on occurrence of the first frame(s), and to check for initial out-of-synchronisation errors. From its initial location *i0*, there are three options: either a sound frame is ready first (signalled through port *fSR*), in this case, the Initialiser sends to the presentation device the order to present the first sound frame (port

fSP), starts the Sound Manager (port *IS*) and the Skew Observer (port *ISk*) and a timer x_i, and moves to location *sf1*. If the first video frame is ready (signalled through *fVR*) before the timer reaches 150 (maximum allowed time for video lagging behind), it sends to the presentation device the order to present the first video frame (port *fVP*), and starts the Video Manager (*IV*) and the Jitter Observer (port *IJ*). If x_i reaches 150 and no video frame is ready (initial out-of-synchronisation error), the Initialiser moves to its error location *iE*. In case a video frame arrives first, the behaviour is (mostly) symmetric: the Initialiser moves to location *vf1*, starts the Video Manager and the Jitter Observer, and waits for the first sound frame to be ready. The timeout in this case is 15, which is the maximum amount of time by which the video presentation may precede the sound presentation. The third option from the initial location is the case where both frames (video and sound) arrive at the same time. In this case, all communication (presentation of frames, initialisation of other components) takes place at the same time, and no further check for initial out-of-synchronisation errors is required.

5.3.2.6 Verification

We consider the behaviour of the LSP in different environments, i.e., with different streams. As explained above, the LSP is designed in such a way that no jitter is allowed on the Sound Stream. Consequently, every Sound Stream that allows jitter would immediately cause an out-of-synchronisation error. We therefore only consider a model of an ideal Sound Stream, as shown in Figure 5.30. Every 30ms, the Sound Stream emits a signal that a sound frame is ready, through port *fSR* for the first sound frame, and through port *SR* for all subsequent frames. The extra port for the first sound frame is introduced to ease proper initialisation of the protocol (cf. the Initialiser in Section 5.3.2.5).

For the Video Stream, we consider two different variants, cf. [BFK$^+$98]. Figure 5.31 shows a Video Stream with non-anchored jitter of 5ms, i.e., every video frame is ready at least 35ms and at most 45ms after the previous frame. As a second variant, Figure 5.32 shows a Video Stream with anchored jitter of 5ms, i.e., every video frame is send not earlier than 5ms before its ideal presentation time, and not later than 5ms after its ideal presentation time. As for the Sound Stream, the signal indicating that the first video frame is ready is sent through port *fVR*, for all subsequent frames, it is sent through port *VR*. The TCA modelling the entire system (including the environment) is obtained by composing the TCA T_{LSP} (cf. Page 110) with the TCA of the Sound Stream (Figure 5.30) and the TCA of the Video Stream with Anchored Jitter (5.32) or Non-Anchored Jitter (Figure 5.31). For brevity of explanation, we may identify the system by the type of jitter of the included Video Stream, for example, we may refer to the fact that "a property holds in the system including the Video Stream with Anchored Jitter" by simply saying "the property holds in the system with anchored jitter" or "the property holds under anchored jitter".

To identify out-of-synchronisation errors, we check for reachability of the error locations *sE* in the Sound Manager, *jE* in the Jitter Observer, and *skE* in the Skew Observer component. Note that instead of the latter two, we could also check

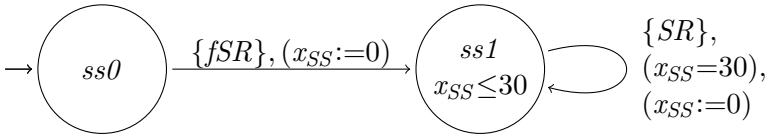

Figure 5.30: LSP, Sound Stream

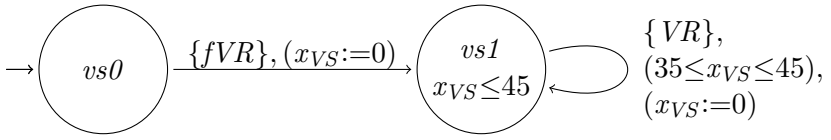

Figure 5.31: LSP, Video Stream, Non-Anchored Jitter

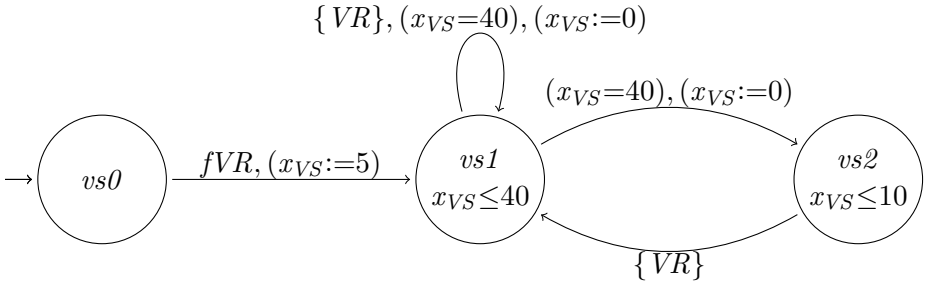

Figure 5.32: LSP, Video Stream, Anchored Jitter

whether the error location *vmE* in the Video Manager component is reachable, since *vmE* is only reachable if either *jE* or *skE* is reachable at the same time, cf. Sections 5.3.2.3 and 5.3.2.4. However, the results would be less meaningful in that case, since we would not be able to identify the exact reason of the error.

For the verification, we further assume that the streams start at the same time, that means the first frames on the two streams are ready at the same time. This is implemented by restricting the runs of the system to those where the Initialiser takes the transition from its initial location *i0* to location *sv* (cf. Section 5.3.2.5). Without this assumption, there can be an arbitrary delay between the first frames on the two streams, and initial out-of-synchronisation errors are trivially possible.

We have modelled the TCA of the LSP with the ECT plugin. As for the ABP (cf. Section 5.3.1.1), we compare three unfolding depths $k \in \{20, 50, 100\}$. Table 5.33 shows the experimental results for the different unfolding depths, error locations and Video Streams (anchored or non-anchored jitter).[13] Again, slanted entries (together with the entry *(S)*) indicate that the input problem is satisfiable. The LSP is modelled in a very concise way already, therefore, we do not provide an abstraction function. In particular, there exists no obvious abstraction function (and probably there exists no non-empty abstraction function at all) that would preserve the results

[13]Note that for k=100, we have added information to the input problem that was obtained from the case k=50, namely that locations *sE* and *skE* are not reachable within 50 steps, and that location *jE* is not reachable within 50 steps under non-anchored jitter. Without this assumption, verification would take much longer for k=100, for example more than 10 hours to detect that location *skE* is not reachable under anchored jitter.

in Table 5.33.

	k=20		k=50		k=100	
	anc.	non-anc.	anc.	non-anc.	anc.	non-anc.
sE	1.145s 22.02MB	0.584s 21.14MB	13.46s 36.77MB	4.171s 32.79MB	43.15s 69.79MB	20.99s 67.22MB
jE	*1.225s (S)* *21.91MB*	1.153s 21.67MB	*3.881s (S)* *32.98MB*	2.075s 31.31MB	*92.27s (S)* *79.06MB*	73.75s 79.90MB
skE	5.250s 22.66MB	4.401s 22.56MB	9.795s 35.22MB	7.769s 33.26MB	1155s 142.44MB	*781.5s (S)* *140.05MB*

All experiments have been carried out with MathSAT, version 4.2.17, on an Intel Core2 Duo CPU E4500, with 2.20GHz and 2.5GB RAM

Table 5.33: Experimental Results for the LSP

The first row in Table 5.33 shows that the error location *sE* in the Sound Manager is unreachable under all unfolding depths and for both types of jitter, that means sound frames can never arrive late. This is the expected result, since we have assumed an ideal Sound Stream in Figure 5.30, which does not experience any jitter.

Out-of-synchronisation errors on the Video Stream (jitter) can occur under anchored jitter only, as shown in the second row of Table 5.33. The reason is that under anchored jitter, the maximal distance of two subsequent video frames is 50ms, which is 5ms more than what is allowed by the protocol. The error can occur in case a video frame arrives as early as possible (namely 5ms before its ideal presentation time), and the following video frame arrives as late as possible (namely 5ms after its ideal presentation time). Under non-anchored jitter however, this error is not possible.

In contrast, out-of-synchronisation error between the two streams (skew) can occur under non-anchored jitter only. The reason is that only in this case, the presentation of the video frames can "drift off", that means deviations from the optimal presentation time can sum up, while under anchored jitter, this is not possible. Remember that we allow the video presentation to lag behind by at most 150ms (cf. the beginning of Section 5.3.2). Assuming all video frames arrive as late as possible, i.e., after 45ms, the first time this error can occur is when the protocol has run for 1395ms: after 1350ms, video presentation lags behind by exactly 150ms, and on arrival of the next video frame 45ms later, the limit of 150ms is exceeded. Due to the periodic nature of the Sound Stream and the Video Stream, the protocol needs more than 80 steps to reach this point, which is why the error occurs under unfolding depth 100 only.

As for performance, we make the following observations: comparing the verification times for reachability of different locations, but with the same unfolding depth and Video Streams, it can be seen that checking the reachability of *sE* is mostly faster than checking the reachability of *jE*, which in turn is faster than checking

the reachability of *skE*. The reason lies in the complexity of the TCA, and the formula representation generated for them. The transition relation representation (cf. (3.13)) of the Sound Manager contains three elements for source location *s1* (two elements of type (3.11) corresponding to the two visible transitions in location *s1*, and one element of type (3.17) for the delay in location *s1*). For the Jitter Observer, the transition relation representation contains five elements for source location *j1* (four elements of type (3.11), and one element of type (3.17)), and for the Skew Observer, the transition relation representation contains eight elements for source location *sk1* (six elements of type (3.11), one element of type (3.12), and one element of type (3.17)). In particular, the Skew Observer has an invisible transition, which can be executed independently of other TCA (subject to the clock guard being satisfied). Intuitively, this means that the transition relation representation of the Skew Observer is "more nondeterministic" than the transition relation representation of the Jitter Observer, which in turn is more nondeterministic than the transition relation representation of the Sound Manager. Thus, when searching for a system run ending in the respective error location, the MathSAT solver needs to try more options for the Jitter Observer than for the Sound Manager, and in turn more options for the Skew Observer than for the Jitter Observer.

A second observation is the fact that in case the property is satisfied for a certain unfolding depth (i.e., the error location is unreachable) for both types of Video Streams, verification of the system with anchored jitter still takes up to factor 3 longer than verification of the system with non-anchored jitter, though the number of possible executions of a given length in both systems is the same. Again, the reason lies in the complexity of the TCA and the formula representation of the transition relation. For the Video Stream with Non-Anchored Jitter, the transition relation representation contains two elements for source location *vs1*, while for the Video Stream with Anchored Jitter, it contains three element for source location *vs1*, and two more for source location *vs2*.

In contrast to the ABP, where reachable error locations where found comparably fast even for k=100 (less than four seconds, cf. Table 5.21), it takes MathSAT more than 13 minutes to detect that location *skE* is reachable for k=100 (under non-anchored jitter). Equivalently, it takes MathSAT about one and a half minutes to detect that location *jE* is reachable for k=100 (under anchored jitter), which is comparably long given the fact that reachability of location *jE* is detected in a bit more than a second for k=20. The reason is that the LSP is considerably more complex than the ABP. Moreover, even though *jE* is reachable very quickly (in actually less than ten steps), MathSAT still needs to find a valid execution for the remaining steps when checking the system with k=100.

This last point shows the benefits of starting the verification process with comparably small unfolding depths, and successively increasing the bound (cf. Section 3.2). In case the error location is k-step reachable already for small values of k, there is no need to consider larger unfolding depths, since this might take disproportionately longer (in Table 5.33, we have added the results for location *jE* under anchored jitter for k=50 and k=100 to illustrate exactly this point). If, instead, the error location is not k-step reachable for small values of k, this information can be used to reduce verification times in subsequent checks for k'-step reachability (for $k'>k$).

5.3.3 Advantages of using TCA

The case studies presented above have highlighted the advantages of using TCA in the development process. In general, TCA have—just like other automata-based models—a rather shallow learning curve, compared to other formal models like for example process algebra. This is thanks to the intuitive graphic approach, and the state-transition notion in the drawings that closely corresponds to mental models of such systems. This allows for an easy entry into the subject even for unexperienced users, who can quickly design simple models with TCA. One of the main advantages of TCA over other automata-based models is that they allow for true concurrency, while combining the notions of time and data. Thus, once the designer gets accustomed to the formalism, TCA offer a powerful formalism to design concurrent distributed systems.

Comparing our TCA-based models of the ABP and the LSP to models of the protocols found in the literature further illustrates the modelling power of TCA. The model of the ABP presented in [Mil89] is based on the calculus of communicating system (CCS) [Mil82]. The model does not use concrete data values, but only works with the alternating bits. Moreover, the timing for resending of messages is modelled with a timer that only sends a timeout at "some point" after being activated, but the model does not use concrete time values. Our approach allows for concrete time and data values, and therefore provides a more faithful model of the ABP. The process-algebraic model of the ABP presented in [Fok00] handles different data values, but does not include time at all. Instead, components are assumed to emit messages and acknowledgements with bit b periodically until they receive the next message or acknowledgement with bit $\neg b$. Our TCA model provides a more flexible approach, and can be adapted to different environments (where channels have different delays, for example), in that it allows to specify different delays for the resending of the different messages and acknowledgements.

The model of the LSP presented in [BFK+98] is based on TA, and therefore enjoys the general advantages of automata-based models described above. However, we observe a number of advantages when using TCA instead of TA to model the LSP. Remember that TA do not allow for true concurrency, that means only a single event can be executed in every step. As a consequence, the model in [BFK+98] needs a large number of so-called *committed locations*[14] (large compared to the total number of locations) to model the synchronous execution of a set of actions. This becomes particularly evident when considering the initialising component (called "Sync") in [BFK+98]: it is less powerful than the Initialiser presented here (cf. Section 5.3.2.5), in that it does not check for initial out-of-synchronisation errors, yet the "Sync" component needs 11 locations to ensure certain actions happen at the same time, while the Initialiser needs only seven. Even more, the model in [BFK+98] needs two additional helper components "Video Manager" and "Sound Manager" (not to be confused with our Sound Manager and Video Manager), whose only purpose is to ensure that certain (sequences of) events in different automata are executed at the

[14]The notion of *committed location* is taken from UPPAAL [upp], which is the solver used for protocol verification in [BFK+98]. A committed location in a TA must be left immediately after it has been entered, without delay or interleaving of other (instantaneous) actions.

same time. In contrast, with TCA, we simply put all events that are to be executed at the same time on a single transition.

To detect a negative delay on the Video Stream (that means, video lagging behind), [BFK+98] uses a helper component "Sound Clock". This component models a "backward running" clock, by decreasing a variable by one every time the value of its clock has increased by one. As a consequence, the maximal delay of the system between two subsequent steps is 1, which results in very long execution traces. In our model, we use a memory cell to keep track of the negative delay. This considerable increases the "step length" of the system, since we only need to update the memory cell on optimal presentation times of video frames (i.e, every 40 time units), and on actual presentation of a video frame (cf. Figure 5.26 and Section 5.3.2.4).