LTL Guided Planning: Revisiting Automatic Tool Composition in ETI

Tiziana Margaria
Chair of Service and Software Engineering
University of Potsdam
August-Bebel-Str. 89, 14482 Potsdam, Germany
margaria@cs.uni-potsdam.de

Bernhard Steffen
Chair of Programming Systems
University of Dortmund
Otto-Hahn-Str. 14, 44227 Dortmund, Germany
steffen@cs.uni-dortmund.de

Abstract

We revisit the automatic tool composition feature of the Electronic Tool Integration platform under the perspective of planning. It turns out that in todays terminology, ETIs temporal logic-based synthesis of tool sequences is a form of planning-based automated orchestration. In contrast to AI-based planning approaches, our synthesis approach is not restricted to compute one solution, but it may compute all (shortest/minimal) solutions, with the intent to provide maximum insight into the potential design space.

1 Introduction

Already 10 years ago, the Electronic Tool Integration platform (ETI) associated to Springer's Journal STTT offered an online portal that provided interactive experimentation with and service-based coordination of heterogeneous tools. Coordination meant in modern terms service composition. In particular it provided a whole spectrum of coordination and evaluation capabilities based on the METAFrame [**?**] open tool coordination environment [**?**]. Complex combinations of functionalities taken from different tools could be (semi-) automatically or interactively constructed and tested by online *meta-programming* in a simple, domain-level specification language tailored for loose specification. Also the burden of data format conversion needed to ensure tool interoperability was automatically taken care of within the underlying ETI platform and hidden from the users. Taken together, these features offered an evaluation and coordination support even for application experts with no programming experience.

At that time, it was the only platform offering this kind of hands-on facilities and coordination support, in particular on a non-in-house tools basis. It supported ETI users with an advanced, personalized Online Service that provided systematic orientation, experimentation, and combination of all the tool functionalities integrated into the ETI repository.

The tool coordination was drastically simplified, freed from any programming and technicalities, so that little or no specific knowledge was prerequisite to the use of ETI as a coordination environment. To this aim, ETI provided high-level task specification languages, graphical support for specifications and user interaction, as well as automatic support of the coordination activity by means of automatic synthesis of complex workflows and prototype animation.

In todays' words, it turns out that ETI provided a model driven, service-oriented platform for the user-side orchestration of complex services (in the ETI application domain, complex verification processes that used heterogeneous algorithms and mediators provening from different verification toolsets). In particular, it supported [**?**]

1. automatic, declarative service discovery based on ontologies (implemented as multifaceted taxonomies accessible and browsable via a hypertext system)
2. model based, graphical service orchestration, analogous to modern BPEL editors (based on an own executable orchestration language called HLL),
3. an execution engine for the orchestrated services (based on an HLL interpeter)
4. formal verification of the complex services, since the graphical orchestration directly defined a coordination graph, analyzable via model checking. This facility comprises model checking-based compliance and conformance checking according to policies expressed as temporal logic properties.
5. automatic synthesis of sequential orchestrations and of mediators (data or functionality mediation) via a synthesis algorithm for a semantic variant of linear time logic (SLTL). This facility would be seen today as a mediator discoverer and planner.

Since ETI was designed to provide a web-based experimentation platform with remote tools, the tool functionalities are in fact remote services accessible via the internet,

and the coordination tasks are orchestration/chaoreography specifications for their automated composition.

The five sections following the introduction of the application example describe these five contributions of ETI in terms of the modern, service-oriented terminology just listed. Subsequently Sect. 9 summarizes our conclusions so far and issues to be dealt with for the future development.

2 Application Example: Orchestrating CADP Activities as ETI Services

Tab. 1 summarizes a small excerpt of the basic services offered by the CADP tool suite [7]. It is the same example used in [18] to illustrate an ETI user session, and it is still valid today: the CADP toolsuite evolved and matured over the past decade to become a reference environment for the verification of distributed processes [4]. CADP is one of the tools that are being integrated right now in the new, Web service-based jETI environment that constitutes the technical platform for the jETI-FMICS community [9].

FMICS, the ERCIM Working Group on Formal Methods for Industrial Critical Systems (FMICS) [8], transfers and promotes the use formal methods technology in industry. The ongoing Verified Software Repository Grand Challenge [11] offers a great opportunity to reach this goal, resulting in a more robust and solid software industry in Europe. The FMICS-jETI platform [9] is a collaborative demonstrator based on the jETI technology [14, **?**]. It provides as repository a collection of verification tools stemming from the activities of the FMICS working group and facilities to orchestrate them in a remote and simple way [1, **?**]. At the same time FMICS-jETI itself is a contribution to the VSR repository and thus to the Grand Challenge.

We are going to illustrate the ETI working style on the same example used 10 years ago to introduce ETI.

3 Ontology-based Service Discovery and Browsing

As in modern SO environments, the service description, management, and retrieval was based on an abstract classification according to behavioural and interfacing criteria. Interfacing criteria correspond to modern WSDL descriptions, while behavioural descriptions correspond to semantic annotations, similar to the modern SAWSDL style. In modern terms, we use an ontology as specification of a conceptualization. In ETI we applied the same principle (see Tab. 1): each CADP tool functionality was provided as a basic ETI service, and abstractly described and published as an *activity* that transforms a (typed) input into the corresponding (typed) output. This way, each functionality offered by the CÆSAR/ALDÉBARAN Toolset [7] toolsuite corresponded directly to an activity in ETI, as shown in Table 1, and was equipped with a conceptualization that linked the interface (static, view as a component) and behavioural aspects (dynamic, view as a function or as a transformer), and their relation within an abstract conceptualization scheme.

This uniform view on services is essential for interdomain applications, one of the major goals of the STTT/ETI venture: it allows us to reuse functionalities across application domains simply by modifying or extending their abstract semantic description (Sect. 3.1).

Both the interfacing and the behavioural descriptions were structured and presented in ETI by means of what we called multifaceted, multicriterial *taxonomies*, which is a common way of implementing ontologies [**?**].

3.1 The ETI Taxonomies

The collection of available functionalities is taxonomically classified for ease of retrieval according to behavioural and interfacing criteria in the activity and type taxonomy [**?**] respectively. Since the ETI taxonomies can be accessed and browsed online, as shown in Fig.1, it is easy to explore the platform's repository, and to orient oneself even in a new application domain.
Activities and types have an associated abstract description in terms of a *taxonomic classification* which establishes their (coarse) application profile. Far from capturing the complete semantics of the underlying tool functionalities (like e.g. algebraic specification approaches), taxonomic specifications are intended to provide abstract, application-specific characteristics based on a collection of predicates.

Formally, taxonomies are directed acyclic graphs, where sinks represent concrete activities/types, the atomic entities of the taxonomy, and where intermediate nodes represent groups, i.e., sets of activities/types with a particular profile. Edges reflect an *is-a* relation between their target and source node. E.g. in the full ETI type taxonomy shown in Fig. 1(top), the `Graph` type is an abstract characterization standing for any of the `CADPGraph`, `RealTimeGraph` types, or equivalently, each type in {`CADPGraph`, `RealTimeGraph`}, like for example the concrete CADP `AUTFile` format, *is-a* `Graph` type.

The same classification principle applies to functionalities too. The portion of the ETI activity taxonomy of Fig. 1 illustrates in the pure CADP view of the ontology for the portions contributed by the CÆSAR/ALDÉBARAN Toolset [7]. In fact, all the types and functionalities integrated in the ETI are classified in exactly this fashion.

The type and activity taxonomies are of vital importance for users' orientation within the data formats and functionalities offered in ETI. As shown in Fig. 1, they are easily reachable from the Toolbar: the Show Activity Taxonomy

activity name	input type	output type	description
Caesar			compiles and verifies LOTOS specifications
caesarAUT	*LOTOSFile*	*AUTFile*	transforms LOTOS programs into LTS in the .aut format
caesarBCG	*LOTOSFile*	*BCGFile*	same as above, but output in BCG format
Aldebaran			minimizes and compares LTS
aldebaranDET	*AUTFile*	*AUTFile*	determinizes an LTS given in .aut format
aldebaranDEAD	*AUTFile*	*CADPStateSet*	computes the deadlock states of an LTS in .aut format
aldebaranLIVE	*AUTFile*	*CADPStateSet*	same, for livelock states (tau circuits)
aldebaranMIN_STD_B	*AUTFile*	*AUTFile*	minimizes an LTS wrt. strong bisimulation equivalence
aldebaranMIN_STD_I	*AUTFile*	*AUTFile*	minimizes an LTS wrt. $\tau^* a$ bisimulation equivalence
aldebaranMIN_STD_O	*AUTFile*	*AUTFile*	minimizes an LTS wrt. observational equivalence
exp2aut	*EXPFile*	*AUTFile*	computes the LTS of a network of communicating LTS
Bcg_io			converts graphs from and into the BCG format
autF2bcgF	*AUTFile*	*BCGFile*	transforms files in .aut format into BCG format
bcgF2autF	*BCGFile*	*AUTFile*	transforms files in BCG format into .aut format
Bcg_open			executes OPEN/CAESAR applic. programs on BCG graphs
bcgTERM	*BCGFile*	*CADPStateSet*	Terminator: a deadlock detection tool
bcgEXEC_RANDOM	*BCGFile*	*SEQFile*	Executor: produces a random exec. sequence on an LTS
...	...	...	...

Table 1. Some of the CÆSAR/ALDÉBARAN Activities

and Show Type Taxonomy commands let ETI load the respective graphical representations in separate windows.

The Type Taxonomy is partially shown in Fig. 1 (top). Here we see the concrete realization of the (application domain independent) distinction between Graph types and File types, and their further, application-oriented subdivision. The type ETIInfo is the generic ETI standard output.

The Activity Taxonomy portion of Fig. 1 (middle) illustrates the classification policy for the tools and functionalities available in the platform. The combinable functionalities are the basis for the coordination. They are classified at the level of single activities. Already in this snapshot we see that classification criteria can be behavioural (e.g. interfaces), application dependent (e.g. tool_activities), and architectural (e.g. cadp_activities), allowing multi-dimensional semantic characterizations of the activities.

The classification scheme depends on the needs of the whole ETI repository, and evolves with the addition of new tools and application domains.

Developing adequate taxonomies is a crucial part of domain modelling within the ETI instantiation process. Exactly as in ontologies, the automatic orchestration synthesis component uses the taxonomies for discovery and mediation. Thus it relies on an appropriate classification of the activities according to their role in the considered domain, in order to provide users with an application-specific handle to the integrated functionalities. The taxonomies must be extended not only when new functionalities are integrated, but also whenever one wants to establish an 'application-specific view' on the so far integrated functionalities: the same tool may well be taxonomically classified completely differently in different application domains. E.g. a compiler may be regarded just as a transformer in one application area, whereas it may constitute the central component in another application area. The corresponding extensions of the 'access language' are essential for inter-domain applications and thus for the technology transfer characterizing STTT's goals: they allow in fact the exchange of functionalities between application domains in a 'symbolic' fashion.

Constructing the taxonomies for this classification can be considered as a meta-integration step, requiring specific knowledge about the considered application domain and its abstraction into *ontologies*.

4 Model Based Service Orchestration

Model based, graphical service orchestration, analogous to modern BPEL editors, was based on HLL, an own executable imperative orchestration language. We called this mode of defining orchestrations *Exact Coordination* [**?**] .

Exact Coordinations could be graphically constructed by drag and drop of activities from the activity taxonomy. Since each activity was also associated at integration time with an executable HLL call, similar to the Web service technology. The resulting orchestrations were displayed as workflows, as shown in Fig.2, within the Service Logic Graph editor (SLG editor) of METAFrame, the application

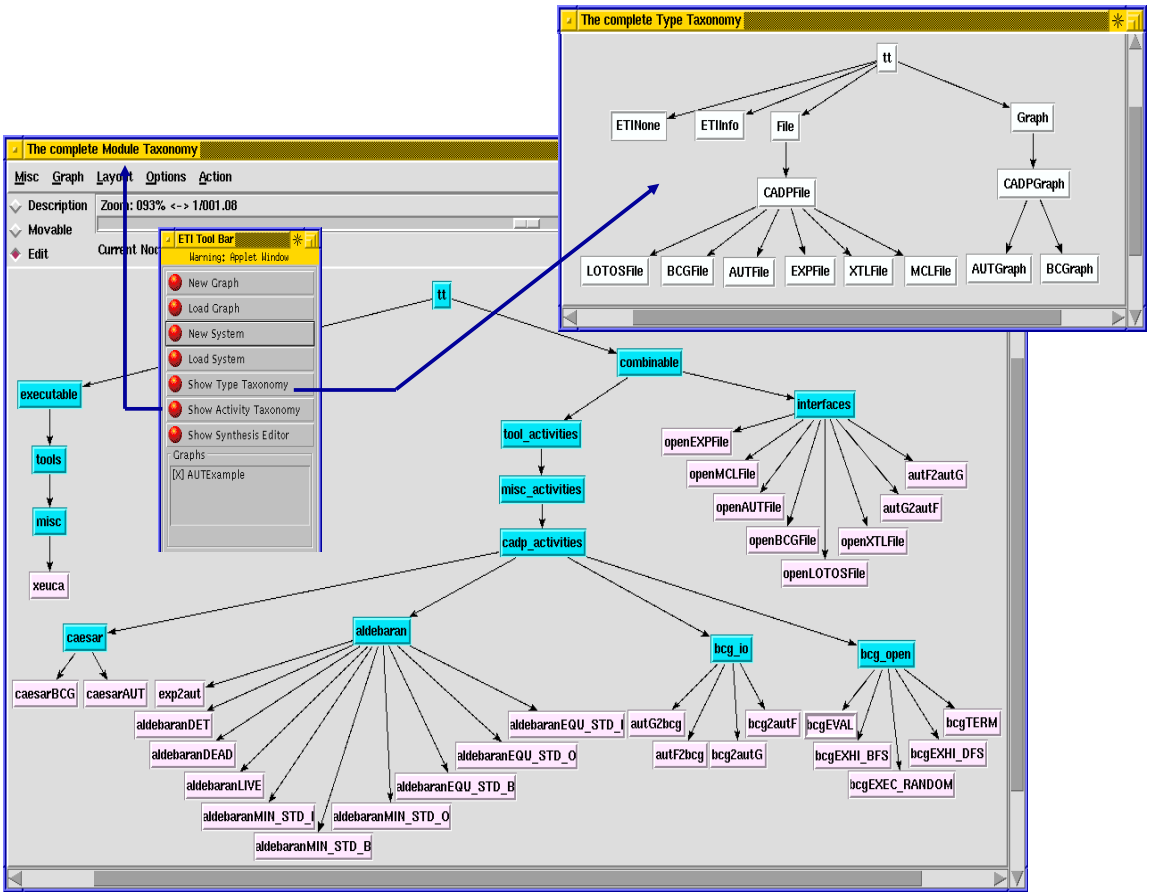

Figure 1. The CADP Type and Activity Taxonomies

development environment underlying ETI. SLGs were interpreted in the underlying runtime environment as HLL processes, thus immediately executable.

As for BPEL, next to the graphical orchestration there was a programming-level representation. ETI coordination could be directly programmed in the interpreted, imperative language HLL (High–Level Language) [5]. As extensible, statically typed language, HLL reflects our policy of separation between *integration* and *coordination*. It offers at the ETI user level a *uniform view* on all the tool functionalities, that abstracts from any specific technical details like data formats or invocation modalities. After their integration at the adapter abstraction level, similar to the production of a WSDL description and service invocation, tool functionality invocations are simply HLL function calls.

5 The Execution Engine

In analogy to BPEL engines, HLL programs can be directly executed via the HLL interpreter [**?**, 3] embedded in the ETI environment, bringing (complex) combinations of functionalities to execution. As we see in Figs. 1 and 2, single tools in the taxonomy and coordination sequences are immediately executable by pressing the Exec button. This form of animation and rapid prototyping allows ETI users to immediately experience the potential not only of single tools, but even of tool combinations. New tool coordinations with a new application profile may be 'stuck together' in order to investigate the interplay between different methodologies in the context of one's own applications.

As in modern service oriented execution platforms, any computation takes place within the coordinated tools. The HLL interpreter simply *delegates* the computational aspects to the tool level, via the tool functionality adapters. Accordingly, it differs from conventional interpreters for imperative programming languages in the implementation of the mechanisms dealing with HLL data types, data values and functions. In fact, the HLL interpreter implements a (single) *generic* wrapper for tool data, which is instantiated by each adapter for its own tool data type during delegation.[1]

[1] Such specific instances can be added at runtime, thus enlarging the language 'on-the-fly'.

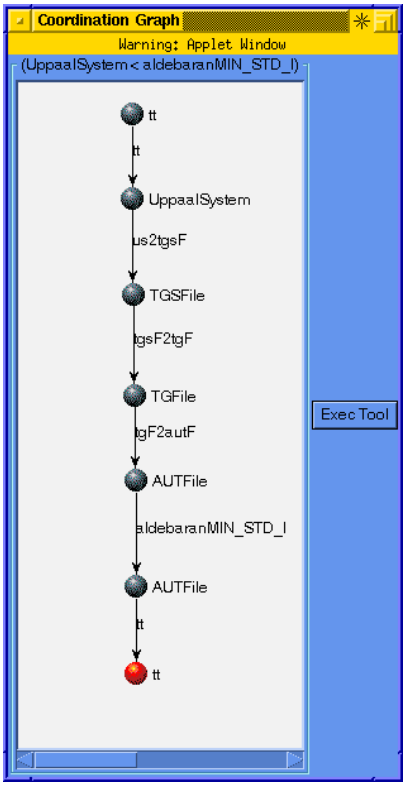

Figure 2. BPEL-style Orchestration: Uppaal and CADP

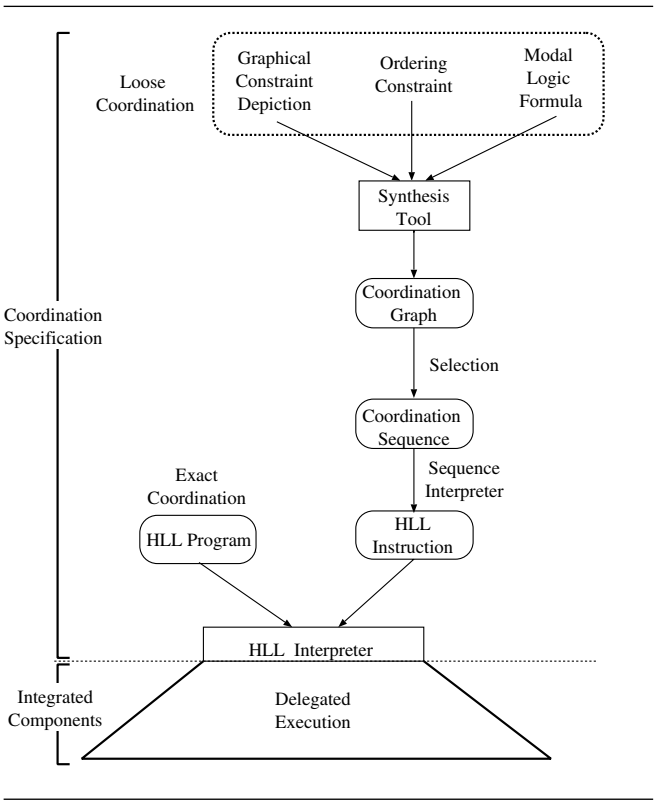

Figure 3. Overview of the Tool Coordination Environment

6 Formal Verification and Compliance

Since ETI orchestration models are in particular SLGs, they can be analyzed with all the means offered by METAFrame [**?**] and its successors, like the jABC [**?**, 13]. Besides very local requirements, which only relate to particular parts of a coordination model, there are also global requirements which are associated with the entire model. These requirements are often very intuitive, are independent of the concrete model, often are part of the rules of the game for an application domain, and can be easily expressed by the application expert.

Model checking is an established approach to automatic verification of systems. It provides an effective way to determine whether a given system is consistent with a specified property. The underlying framework incorporates this technique via a core model checking facility. Intuitively, any system modelled as SLG can be verified with this technique, independently of the concrete interpretation of the SLG. We see for this purpose SLGs as graphs (in particular, as Kripke Transition Systems) consisting of nodes holding labels that the model checker interprets as atomic propositions (for example the type names), and edges annotated with action names that for the model checker represent actions in the specification logic. Fig. 2 shows a simple sequential example of such a coordination model. Properties of such a model can be specified using temporal logics, for example CTL (Computation Tree Logic) or the modal mu-calculus [16]. The user can add and describe properties, check properties for a particular model, and interactively investigate the error diagnosis information via error views.

The foreseen use was an a posteriori verification of properties for the coordination models resulting from the exact coordination described above, in terms of compliance to certain rules of the game. For example, constraints that express reachability of certain types or activities.

Model checking was all the time present in the ETI environment through the underlying development environment, but it has not been used in practice. It was used extensively in the METAFrame and in the jABC environment, where it was essential part of the Service Development process in the Intelligent Network application [**?**, **?**], and later in the design of automatically executable test suites for Computer-Telephony Integrated systems [10, 12].

The central use of temporal logic in the ETI environment concerned in contrast the automatic generation of orchestration from loose temporal logic specifications, which we discuss in the following.

7 Automatic Synthesis of Sequential Orchestrations

ETI featured the construction of directly executable tool coordination sequences on the basis of 1) the repository of functionalities and transformers, 2) of the taxonomies, and of 3) a loose coordination specification language to express abstract constraints on the desired sequences.

The way the synthesis is built closely resembles a planning engine that solves loosely specified and LTL-guided planning problems.

7.1 Loose Coordination: Specifying Abstract Constraints

Loose coordination languages characterize adequate coordination sequences in terms of abstract constraints. Adequacy of the desired sequences is expressed via a collection of global liveness and safety properties on the activity and type descriptions contained in the taxonomies. Looseness is thus supported in two orthogonal dimensions:

- local characterization of single types, functionalities or transformers at the abstract level of the taxonomies - similar to the specification of a planning problem, and
- whole characterization of whole coordination sequences in terms of abstract constraints specifying precedences, eventuality, and conditional occurrence of single taxonomy objects - the LTL guidance for the planning engine.

Local Constraints: pre- and post assertions and Goals Local constraints express selection criteria for single types and activities on the basis of the respective taxonomic classification. We have

- *Type constraints* describing the 'neighbourhood' of components. Here we consider type compatibility as the only criterion.
- *Activity constraints* characterizing the set of activities which may appear in the solution.

Type and activity constraints are accordingly formulated as simple propositional logic formulas over the respective taxonomies, which are regarded as definitions of *sets of* basic predicates (atoms).

Definition 7.1 (Taxonomy expressions)
Let TAX be a taxonomy over some set. Then we can construct the corresponding set of taxonomy expressions by:

$$TE \quad ::= \quad A \mid \neg\, TE \mid TE \;\wedge\; TE \mid TE \;\vee\; TE$$

where nodes $A \in TAX$ are taken as atomic propositions.

The taxonomies are in general directed acyclic graphs, where concepts can be described along different facets. Facets can characterize both the functionality and also non-functional properties of an activity or type. The CADP taxonomies of Fig. 1 are thus trees by coincidence.
As an example, in an activity taxonomy for graphics tools the activity constraint

$$\texttt{filter} \wedge \neg(\texttt{scale} \vee \texttt{reduce})$$

characterizes the set of all `filter` commands in the activity taxonomy that perform neither scaling nor reducing manipulations.

7.2 Temporal Constraints: SLTL

Temporal logics have been successfully introduced in various other areas of computer science to capture global properties of finite state systems. The temporal logics considered here capture

- general ordering properties, like
  this activity/type must be executed/reached some time before another activity,
- abstract liveness properties, like
  a certain activity/type is required to be executed/reached eventually, and
- abstract safety properties, like
  two certain activities/types must never be executed/ reached simultaneously.

This way users can specify elaborate requirements concerning (some of) the activities (tools and transformations) and types constituting a desirable coordination sequence.

ETI's loose coordination specification language was the *Semantic Linear-time Temporal Logic* (SLTL) [**?**, **?**], a temporal (modal) logic comprising taxonomic specifications of types and activities as embedded semantical element.

Definition 7.2 (SLTL)

The syntax of Semantic Linear-time Temporal Logic (SLTL) is given in BNF format by:

$$\Phi ::= \quad \mathsf{type}(t_c) \mid \neg\Phi \mid (\Phi \wedge \Phi) \mid <a_c>\, \Phi \mid \mathbf{G}(\Phi) \mid (\Phi \mathbf{U} \Phi)$$

where t_c and a_c represent type and activity constraints formulated as taxonomy expressions over the respective taxonomies.

SLTL formulas are interpreted over the set of all *legal coordination sequences*, i.e. alternating type correct sequences of types and activities[2], which start and end with types. Intuitively,[3]

- type(t_c) is satisfied by coordination sequences whose first element (a type) satisfies the type constraint t_c.
- Negation $\neg$ and conjunction $\wedge$ are interpreted as usual.
- Next-time operator <> :
  $<a_c>\ \Phi$ is satisfied by coordination sequences whose second element (the first activity) satisfies a_c and whose *continuation*[4] satisfies Φ. In particular, $<tt>\ \Phi$ is satisfied by every coordination sequence whose continuation satisfies Φ.
- Generally operator **G**:
  $\mathbf{G}(\Phi)$ requires that Φ is satisfied for every suffix[5].
- Until operator **U**:
  $(\Phi \mathbf{U} \Psi)$ expresses that the property Φ holds at all type elements of the sequence, until a position is reached where the corresponding continuation satisfies the property Ψ. Note that $\Phi \mathbf{U} \Psi$ guarantees that the property Ψ holds eventually (strong until).

The definitions of continuation and suffix may seem complicated at first. However, thinking in terms of path representations clarifies the situation: a subpath always starts with a node (type) again. Users should not worry about these details: they may simply think in terms of pure activity compositions and not care about the types, unless they explicitely want to specify type constraints.

The online introduction of *derived operators and patterns* supports a modular and intuitive formulation of complex properties. This eases defining the dual operators:

False :	*ff*	$=_{df}$	$\neg tt$
Disjunction :	$\Phi \vee \Psi$	$=_{df}$	$\neg(\neg\Phi \wedge \neg\Psi)$
Box :	$[a_c]\Phi$	$=_{df}$	$\neg <a_c> (\neg\Phi)$
Eventually :	$\mathbf{F}(\Phi)$	$=_{df}$	$\neg\mathbf{G}(\neg\Phi) = (tt\ \mathbf{U}\ \Phi)$

The following two simple examples illustrate typical loose sequencing constraints, conveniently specified in SLTL:

- $\mathbf{F}(< display > tt)$
  which means 'A $display$ activity will occur eventually, i.e. now or at a later point of time'.
- $\mathbf{G}\ (\ (< tex\_modules > tt\ ) \Rightarrow \mathbf{F}(< display > tt)\ )$
  which is a liveness property meaning 'Whenever a $tex\_modules$ activity occurs, then a $display$ activity is guaranteed to eventually occur as well'.

7.3 Synthesis of Coordination Sequences

Once a loose coordination specification in SLTL is entered into the corresponding editor, an *automatic synthesis* tool [2, **?**] generates the satisfying plans (in terms of sequences of concrete HLL commands) according to the process shown in Fig. 3(right).

The solution space resulting from the automatic synthesis may contain quite different alternatives, each corresponding to a different minimal-length sequence of functionalities satisfying the specified (loose) constraints. The potential of alternative solutions is presented as a graph and the HLL code associated with each path of this graph can be executed immediately in response to a selection by means of simple mouse clicks.

This way, ETI's coordination environment eliminates any traditional programming: from the loose specification of a coordination task to its execution, users are able to work without any knowledge of the underlying coordination mechanisms, of the coordination language or even of the available tools, which can conveniently be made familiar with via taxonomy browsing.

8 Orchestration Synthesis in Practice: the CADP Example

Practice has shown that when combining tools and tool functionalities, the typical problem description (the coordination task) is characterized in terms of

- a number of functionalities which must be performed at some time during the planned execution (a set of activities),
- maybe a data format (in our modelling, a type) in which the initial data are available
- maybe a desired format for the output data (again a type),
- maybe some hints on desired sequencing between some of the activities (a partial order).

This description is typically incomplete, in the sense that

- the named activities may not suffice to carry out the task, thus one has to look for additional or missing ones (workflow completion),

[2] During the description of the semantics, types and activities will be called *elements* of the coordination sequence.

[3] A formal definition of the semantics can be found in [**?**].

[4] This continuation is the coordination sequence starting from the third element.

[5] According to the difference between activity and type components, a suffix of a coordination sequence is any subsequence which arises from deleting the first 2n elements (n any natural number).

- the input and output data formats may not be read resp. delivered by the named activities, thus some additional data format conversions (transformations) are necessary (data mediation),
- the given ordering, if any, is not total.

Thus many alternatives may be possible, depending on the way the unspecified information is filled.

Type completion, the simplest form of loose coordination introduced in [**?**], already covers this class of problems. Let us see how to use it on a simple concrete example.

8.1 Type-Incomplete Sequences

We start from the following informal problem description:

> *Given a benchmark system in* LOTOS *format, we want to compute an equivalent minimal system and model check it.*

It is easy to derive a corresponding formal specification: with some navigation in the type and activity taxonomy we learn that systems described in the LOTOS process algebra are represented by the type `LOTOSFile`, that the minimization activities are collectively classified in the taxonomy as minimizer, and that model_checker is the activity group which, as intuitive, groups the model checking tools.

The corresponding (formal) loose specification of the coordination task is input via the *Synthesis Editor* as shown in Fig. 4(left). Using derived operators like < for **precedes** [**?**], we may simply write:
`((LOTOSFile < minimizer) < model_checker)`
This specification is both locally and globally loose:

- locally loose, because `minimizer` does not uniquely determine an activity, and
- globally loose, because the specification is type inconsistent and therefore requires the insertion of intermediate transformers (mediators) in order to arrive at executable solutions.

We wish to display all the *shortest* executable coordination sequences, but ETI provides also other options: all solutions, all minimal solutions or just one shortest solution [**?**].

8.2 Obtaining the Coordination Sequence

Pressing the Submit button, the specification is sent to the automatic synthesis tool and a coordination graph containing the desired coordination sequences is displayed in a separate window.

Fig. 4(middle) shows that there are 12 shortest solutions, graphically presented as paths of the displayed coordination graph. The length 6 indicates that the problem of type completion is more intricate than one would have expected. A quick analysis of the solutions tells us why.

1. The `LOTOSFile` must first be opened and converted into a corresponding input in CADP format (of type `AUTFile`). This is done via the activity `caesarAUT`, with signature

   ```
   (LOTOSFile, caesarAUT, AUTFile)
   ```

   (see Table 1), which takes a LOTOS program and returns a model in AUT format for the CADP toolset. This activity is clearly a type transformer.

2. Any of the minimizer activities currently available in the activity taxonomy of Fig. 1 is now applicable, since they all accept the input type `AUTFile`, and are therefore compatible. They are all supported by ALDÉBARAN [**?**], have signatures

   ```
   (AUTFile, aldebaranMIN_STD_B, AUTFile)
   (AUTFile, aldebaranMIN_STD_I,AUTFile)
   (AUTFile, aldebaranMIN_STD_O, AUTFile)
   ```

   and differ wrt. the notion of equivalence they implement (strong bisimulation, weak bisimulation, and observational congruence [7]). All three deliver the resulting automaton as a file of type `AUTFile`.

3. The minimized graph must now be converted into the BCGFile format for the model checking. A second transformer `autF2bcgF`, with signature

   ```
   (AUTFile, autF2bcgF, BCGFile)
   ```

   delivers the corresponding file, of type `BCGFile`. This is a special proprietary format (Binary encoded Graph) of CADP, which represents also large models in a compact fashion. It is the common input format for the model checkers of the CADP tool suite.

4. Any of the model_checker activities currently available in the CADP activity taxonomy is now applicable, since they all accept the input type `BCGFile`, and are therefore compatible. Both `bcgEVAL` and the `xtl` model checker are here suitable.

5. Finally, their output can be either stored in a file (of generic type `ETIFile`, since different) via the activity `saveETIFile`, or textually displayed on the screen via the activity `showTextFile`, where `ETIResult` is the final abstract type in ETI.

ETI's conversion mechanism takes care of all this completely automatically. In fact, ETI newcomers may be simply provided with an execution button for the execution of a default coordination sequence. This completely hides the entire synthesis and mediation mechanism.

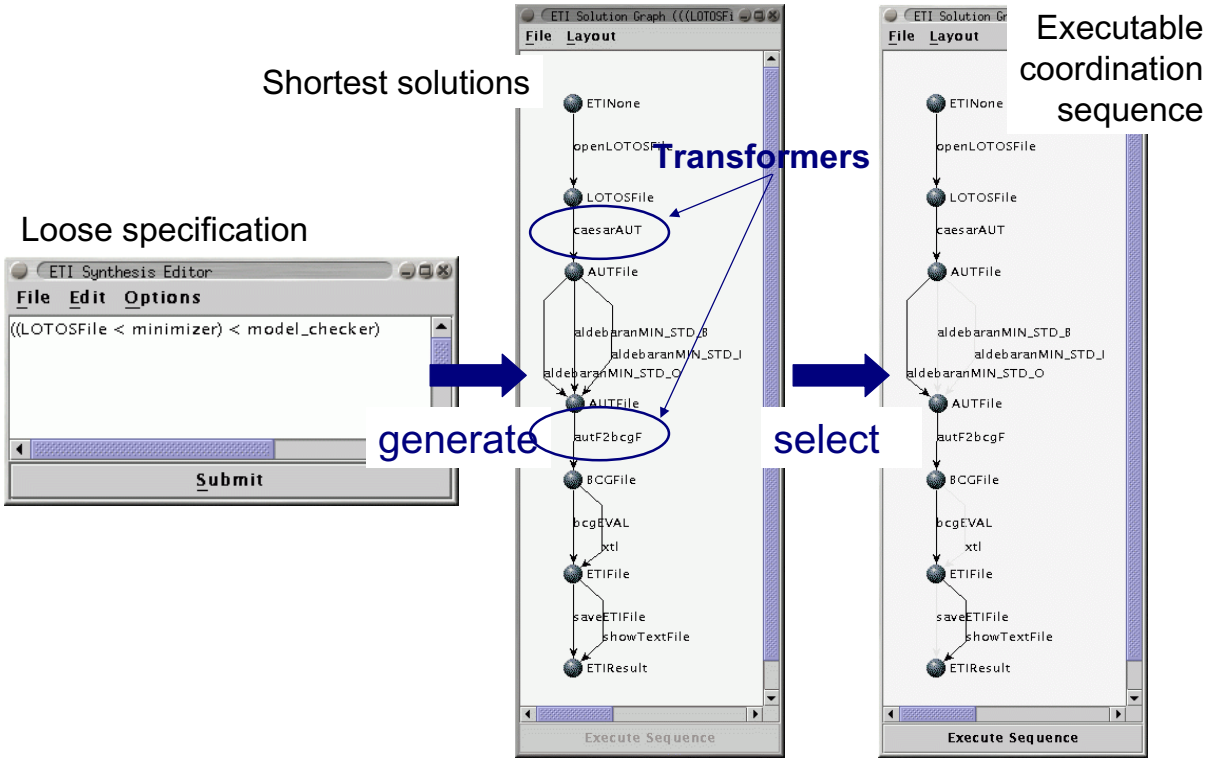

Figure 4. The CADP Resulting Coordination Graph

8.3 Executing a Coordination Sequence

Clicking with the mouse on an edge of a coordination sequence selects all the paths (solutions) to which it belongs. If we think that an appropriate notion of equivalence is observational bisimulation, we just have to click on the edge

```
(AUTFile, aldebaranMIN_STD_O, AUTFile)
```

and similarly select the paths through the `bcgEVAL` model checker and to a display of the resulting file to reach the situation shown in Fig. 4(right): since only one path is now selected, (the highlighted one), the corresponding solution is immediately executable (enabled button on the bottom).

9 Conclusions

We have revisited the automatic tool composition feature of the Electronic Tool Integration platform under the perspective of planning. In contrast to AI-based planning approaches, our LTL-based synthesis approach is not restricted to compute one solution, but it may compute all (shortest/minimal) solutions, with the intent to provide maximum insight into the potential design space.

In future we plan to investigate various forms of synthesis approaches in order to reveal their application profiles. In particular, we are interested in comparing game-based methods which work via the synthesis of winning strategies with tableau-based methods that construct a linear model as a result of proof construction. We also plan to enhance the user-friendliness in terms of graphical support for the specifications, for example by means of the formula Builder [15] and by the use of patterns [6]. We also plan to apply this approach to the mediation problem of the ongoing Semantic Web Service Challenge [**?**, 17].

References

[1] A. Arenas, J. Bicarregui, T. Margaria: *The FMICS View on the Verified Software Repository*, Proc. Integrated Design and Process Technology, IDPT-2006, San Diego (USA), 26-29.6.2006, Society for Design and Process Science, 2006.

[2] M. von der Beeck, B. Steffen, T. Margaria. *A Formal Requirements Engineering Method and an Environment for Specification, Synthesis, and Verification*, Proc. of SEE '97, 8th IEEE Conference on Software Engineering Environments, Cottbus (Germany) 8-9 April 1997.

[3] V. Braun, T. Margaria, C. Weise: *Integrating Tools in the ETI Platform*, Software Tools for Technology Transfer, Vol. 1, Springer Verlag, December 1997.

[4] CADP Website: `www.inrialpes.fr/vasy/cadp/`

[5] A. Claßen, B. Steffen, T. Margaria, V. Braun: *Tool Coordination in* METAFrame, Techn. Rep. MIP-9707, Fakultät für Mathematik und Informatik, Universität Passau, Passau (Germany), April 1997, 25 pp.

[6] M. Dwyer, G. Avrunin, J. Corbett. *Specification Patterns Website*. patterns.projects.cis.ksu.edu/.

[7] J.C. Fernandez, H. Garavel, A. Kerbrat, R. Mateescu, L. Mounier, M. Sighireanu, *Cadp: A protocol validation and verification toolbox*, Proc. CAV'96, New Brunswick, NJ, USA, LNCS, Springer Verlag, August 1996.

[8] FMICS WG homepage: http://www.inrialpes.fr/vasy/fmics/

[9] Homepage of the FMICS-jETI platform: http: jeti.cs.uni-dortmund.de/fmics/index.php.

[10] A. Hagerer, T. Margaria, O. Niese, B. Steffen, G. Brune, H.-D. Ide: *Efficient Regression Testing of CTI-Systems: Testing a Complex Call-Center Solution*, in Annual Review of Communication, Int. Engineering Con-sortium Chicago (USA), Vol. 55, pp.1033–1039, IEC, 2002

[11] C.A.R. Hoare, J. Misra: *Verified software: theories, tools, experiments - Vision of a Grand Challenge project*, July 2005, http://vstte.inf.ethz.ch/pdfs/vstte-hoare-misra.pdf

[12] H. Hungar, T. Margaria, B. Steffen: *Test-Based Model Generation for Legacy Systems*, Proc. IEEE ITC'03, Charlotte, 2003, IEEE CS Press, pp.971–980.

[13] *jABC Website*: http://www.jabc.de.

[14] jETI Website: jabc.cs.uni-dortmund.de/ plugins/jeti_en.html.

[15] S. Jörges, T. Margaria, and B. Steffen. Formulabuilder: A tool for graph-based modelling and generation of formulae. In Proc. *ICSE'06* Shanghai, May 2006.

[16] D. Kozen. Results on the propositional mu-calculus. *Theoretical Computer Science*, 27:333–354, 1983.

[17] C. Kubczak, R. Nagel, T. Margaria, B. Steffen: *The jABC Approach to Mediation and Choreography*, Semantic Web Services Challenge 2006, Phase I and II Workshops, DERI, Stanford University, Palo Alto, March 2006.

[18] T. Margaria, V. Braun, B. Steffen: *Interacting with ETI: A User Session*, Software Tools for Technology Transfer, Vol. 1, Springer Verlag, December 1997.

[19] T. Margaria, C. Kubczak, B. Steffen, S. Naujokat: *The FMICS-jETI Platform: Status and Perspectives* ISoLA 2006, 2nd IEEE-EASST Int. symp. On Leverag-ing Applications of formal methods, verification, and validation, Paphos (CY), 15-19.11.06, pp. 414-418, IEEE CS Press.

[20] T. Margaria, R. Nagel, B. Steffen: *Remote Integration and Coordination of Verification Tools in JETI*, Proc. IEEE ECBS 2005, April 2005, Greenbelt (USA), IEEE CS Press, pp. 431–436.

[21] T. Margaria, B. Steffen: *Backtracking-free Design Planning by Automatic Synthesis in* METAFrame, Proc. FASE'98, Lisbon(P), April 1998, LNCS, Springer Verlag.

[22] T. Margaria, C. Winkler, C. Kubczak, B.Steffen, M. Brambilla, D. C. S. Ceri, E. D. Valle, F. Facca, and C. Tziviskou. The sws mediator with webml/webratio and jabc/jeti: A comparison. In *Proc. ICEIS'07, 9th Int. Conf. on Enterprise Information Systems*, Funchal (P), June 2007.

[23] R. Roszkiewicz: *Getting Started with Controlled Vocabularies, Taxonomies and Thesauri*, The Seybold Report, Analyzing Publishing Technologies, Vol.5, N.16, 2005, Seybold Publications. www.factiva.com/press/ seybold_taxonomy_2006.pdf

[24] B. Steffen, T. Margaria. METAFrame in Practice: Design of Intelligent Network Services. In *Correct System Design - Recent Insights and Advances*, LNCS N. 1710, State-of-the-Art Survey, pp. 390–415. Springer-Verlag, 1999.

[25] B. Steffen, T. Margaria, V. Braun: *The Electronic Tool Integration Platform: Concepts and Design* Software Tools for Technology Transfer, Vol. 1, Springer Verlag, Nov. 1997.

[26] B. Steffen, T. Margaria, A. Claßen, V. Braun, M. Reitenspieß. *An Environment for the Creation of Intelligent Network Services*, in "Intelligent Networks: IN/AIN Technologies, Operations, Services, and Applications – A Comprehensive Report", Int. Engineering Consortium, Chicago IL, 1996, pp. 287-300.

[27] B. Steffen, T. Margaria, A. Claßen, V. Braun, M. Reitenspieß, H. Wendler. *Service Creation: Formal Verification and Abstract Views*, 4th Int. Conf. on Intelligent Networks (ICIN'96), Nov. 1996, Bordeaux pp. 96-101.

[28] B. Steffen, T. Margaria, R. Nagel, S. Jrges, C. Kubczak: *Model-Driven Development with the jABC*, Proc. HVC'06, IBM Haifa Verification Conference, Haifa (Israel), LNCS 4383, Springer Verlag, October 2006.