2013 IEEE 12th International Symposium on Network Computing and Applications

Tempo-toolkit: Tempo to Java Translation Module

Chryssis Georgiou*, Peter M. Musial†, and Christos Ploutarchou*
* University of Cyprus, Nicosia, Cyprus
Email: {chryssis, christos.ploutarchou}@cs.ucy.ac.cy
†MIT-CSAIL, Cambridge, MA, USA
Email: {pmmusial}@csail.mit.edu

***Abstract*—TIOA is a formal language for modeling distributed, concurrent, and timed/untimed systems as collections of interacting state machines, called *Timed Input/Output Automata*. TIOA provide natural mathematical notations for describing systems, their intended properties, and the relationships between their descriptions at varying levels of abstraction. The Tempo toolkit is an implementation of the TIOA language and a suite of tools that supports a range of validation methods for description of systems and their properties, including static analysis, simulation, and machine-checked proofs. The tools are implemented as Eclipse plugins. In this paper we introduce a new plugin of the toolkit, the *Tempo-to-Java* compiler, which automatically translates high-level Tempo specification into executable Java code for various distributed platforms. The translation process is verified to preserve the formal properties of the source specification, hence leading to generated code which is *correct by construction*.**

***Keywords*—Design tools and techniques, Automatic code generation, Distributed programming, Verifiable translation.**

I. Introduction

Motivation. Developing dependable distributed systems for modern computing platforms continues to be challenging. While the availability of distributed middleware makes feasible the construction of systems that run on distributed platforms, ensuring that the resulting systems satisfy specific safety, timing, and fault-tolerance requirements remains problematic. The middleware services used for constructing distributed software are specified informally and without precise guarantees of efficiency, timing, scalability, compositionality, and fault tolerance. Current software-engineering practice limits the specification of such requirements to informal descriptions. When formal specifications are given, they are typically provided only for the system interfaces. (Middleware interface syntax is usually strongly defined, where computational semantics are often defined superficially.) The specification of interfaces alone stops far short of satisfying the needs of users of critical systems. Such systems need to be equipped with precise specifications of their semantics and guaranteed behavior. When a system is built of smaller components, it is important to specify the properties of the system in terms of the properties of its components.

We view formal specification and analysis as valuable tools that should be at the disposal of the developers of distributed systems. However, theoretically sound specifications have a limited impact, unless tools exist that *automatically* transform these specifications from high-level notation to executable code. Only if such tools are formally scrutinized, then the resulting executable code can be deemed as reliable and verifiably correct.

At the core of this work is the Tempo framework and its toolkit [1] developed by VeroModo, Inc. [2]. Tempo is derived from the formal mathematical modeling framework called *Timed Input/Output (I/O) Automata* [3] that is well established in the theoretical distributed computing research community and has been used to specify and reason about a plethora of distributed and concurrent algorithms, e.g., [3]–[8]. Systems in this framework are specified in terms of interacting timed automata. The Tempo language [9] closely matches semantics of the Timed I/O Automata modelling framework and hence it inherits the rich set of capabilities for system modeling and analysis.

The Tempo toolkit contains tools to support analysis of systems such as, the *front-end* that checks syntax and performs static semantic analysis; a *simulator* to explore execution traces for an automaton; a *translation module* to the UPPAAL model-checker [10]; and a *translation module* to the PVS interactive theorem prover [11]. *The reader is welcome to experiment with these tools, but we do not make any claims about them in this work; we only concentrate on the translation module to Java which we introduce in this paper.*

Our Contribution. Our work involves the development and implementation of a strategy for generating distributed executable Java programs from Tempo specifications, and proving that this strategy preserves the correctness of the source specifications, provided these adhere to certain constraints.

Conceptually, our work can be considered a non-trivial extension of the work in [12], [13], where a strategy for transforming un-timed IOA specifications into Java program was developed (more details are given in Section II). Specifically, the main contribution of our work is the incorporation of time (trajectories) into the transformation methodology. Besides some coding effort in doing so, the main challenge was to prove the correctness of the transformation, which needed to adhere to the trajectory axioms (given in Section IV). In fact, en route in proving the transformation's correctness, we proved a time-based theorem (Theorem 1) that we believe it is applicable beyond our work. We also needed to restrict the source specification that allow layering of time, as the time in TIOA is *logical* and hence different from the lower level time of Java. From a programming aspect, we did not extend the code of the (non-time) IOA complier to model time, but instead we wrote the compiler from scratch. The resulting compiler is now modular and well-structured which makes debugging and future extensions an easier task. As an example of the modularity of the code, is the extension of the compiler to support, besides MPI, also TCP communication, which enables the compiler to develop and run programs over LANs and WANs (MPI restricts the execution of programs only in LANs).

978-0-7695-5043-5/13 $26.00 © 2013 IEEE
DOI 10.1109/NCA.2013.17

As a proof-of-concept implementation, we have used the compiler to implement and evaluate several timed distributed algorithms, including a distributed clock synchronization algorithm we use here as a *running example*. (Details on all implemented algorithms, including Paxos [6], can be found in the technical report [14].) Our translation tool has been successfully used and tested in the classroom setting at MIT and the University of Cyprus, both at undergraduate and graduate level.

Document Organization. In Sect. II we provide necessary background and briefly describe related work. In Sect. III we describe our translation method. In Sect. IV we overview the correctness of the translator; we argue that the resulting Java code preserves the correctness of the source specification. In Sect. V we use the translator to generate two implementations of the running example, where one uses MPI and the other TCP for communication. We conclude in Sect. VI.

II. Background

As mentioned, our work is based on the Timed Input/Output Automata or TIOA [3] framework. Even though a user might be completely unfamiliar with this framework, it is relatively easy to learn and use it, within only a few weeks (cf. [15]). Below we provide information which should be sufficient for one to follow the ideas behind the presented translation module (of course, the more familiar the reader is with TIOA the easier to follow these ideas).

The TIOA framework provides a mathematical basis for modeling and reasoning about distributed systems (both timed and untimed). The flexibility and the power of this framework contribute as follows: (i) The system designer has the flexibility of using *nondeterminism* to allow multiple correct specifications of its system, hence relaxing assumptions on behaviors of the environment in which the system operates (at least at certain parts of the execution). (ii) Complex systems can be decomposed into subsystems, where *composition* of these subsystems yields the unified complex system. Such structured design enables one to view specifications at multiple levels of abstraction.

TIOA model behavior of systems of interacting components (i.e., automata), where system components operate in discrete steps, and timing-related components whose behavior includes continuous transformation over time. For timed components, when the transformation reaches its stopping condition, the time, which in TIOA is simply a set of real numbers where the time progresses over these, stops to allow one or more discrete interactions which are assumed to be instantaneous. Hence, in TIOA the notion of time is a *logical abstraction* and not physical; the fact that time is abstract and can stop is a clear departure from the standard meaning of time, and it needs to be handled carefully when deriving executable code (handling time was the main challenge of our work).

More formally, a Timed I/O Automaton is a labeled state transition system. It consists of a (possibly infinite) set of states (including a non-empty subset of start states); a set of discrete actions, classified as *input*, *output*, or *internal*; a transition relation, consisting of a set of $(state, action, state)$ triples (these specify the effects of the discrete actions); and a set of *trajectories* describing state evolution over time. Specifically, a trajectory is a continuous or discontinuous function that describes the evolution of the state variables over interval of times. An action is enabled if its preconditions are satisfied; input actions are always enabled. TIOA support the *composition operation* by which TIOAs modeling individual timed system components can be combined to produce a model for a larger timed system.

Prior development. The Input/Output Automata (IOA) framework [4] and its *IOA toolkit* [16] is the predecessor of the TIOA framework and allows modeling of *untimed* systems only. Using the IOA notation, a wide range of systems can be specified and reasoned about and the IOA toolkit supports the design, development, testing, and formal verification of concurrent untimed systems. The IOA toolkit is no longer supported and it is not easily expandable to support the development of new tools. Despite its shortcomings, the theoretical work developed during creation of the IOA toolkit is instrumental to the activities of this work. Specifically, it defines the limitations on the kind of specifications that are allowed for compilation to executable code. These limitations apply in our work as well; however, as we discuss later, further restrictions are imposed by our work for handling time.

The pragmatic benefits of transitioning from IOA to Tempo are as follows: *Syntactically* the IOA notation evolved (into Tempo notation) and its syntax is now more refined and standardized. *Implementation-wise* the key difference is that Tempo uses a flexible design that takes advantage of standard technologies, such as the ANTLR grammar parser, abstract syntax tree representation, Java's visitor pattern, and plugin support. *Aesthetically* users will enjoy a nice Tempo interface built atop the familiar *Eclipse Rich Client Platform*. The theoretical contribution of this work, as already mentioned, is incorporating time (i.e., trajectories) into the translation methodology. This presents also some coding challenge (e.g., stopping trajectories across all components), but more importantly, it significantly complicates correctness reasoning of the translation.

Related work. The CCS process algebra [17] and IOA frameworks share several similarities as well as many differences: the work in [18] presents a semantic-based comparison of the two frameworks. The Concurrency Factory [19] is a toolkit for the analysis of finite-state concurrent systems specified as CCS expressions; it includes support for verification, simulation, and compilation. The compilation tool translates specifications into Facile code. The Nuprl proof development system [20] and its toolkit [21] present another relevant body of work. In [22] a framework consisting of Nuprl, IOA, and the OCaml programing model have been used to verify certain properties of Ensemble [8], a group communication system and its implementation. Also the Nuprl toolkit supports translation modules to both Java and OCaml. A contribution of [22] was its formal approach to inheritance and its semantics. In contrast, we place TIOA at the center and use its notation to model systems and its mathematical support for verification. The Tempo toolkit implements a software framework around TIOA with native analysis and translation tools and integration of external verification environments.

Times [23] is a prototype C-code generator based on the legOS operating system applied to develop the control software for a production cell for models specified in Timed Automata [24]. It is appropriate for systems that can be

described as a set of tasks which are triggered periodically or sporadically by time or external events. The toolkit produces software with predictable timing behaviours, i.e. code which is known a priori to satisfy given timing constraints. To our knowledge, the Times compiler does not support the generation of distributed code (e.g., it does not support the integration of a communication medium such as MPI or TCP).

III. From Tempo Specification to Java Code

We now detail how one, starting from a distributed algorithm, can use our translation module to obtain executable java code, while preserving the correctness of the algorithm. As a running example we use a simple Clock Synchronization algorithm as specified in [3], which we overview next.

Clock Synchronization. The TIOA specification of a process i is given in Listing 1 as taken from [3, Section 4.1]. Each process has a physical clock ($physclock$) and generates a logical clock ($logclock$). The goal of the algorithm is to achieve *agreement* and *validity* among the logical clock values; agreement means that the logical clocks are close to one another and validity that the logical clocks are within the range of the physical clocks. The algorithm is based on the exchange of physical clock values between the processes. Parameter u determines the frequency of sending messages and i is the index of the process. State variable $physclock$ may drift from the real time with rate bounded by r. The state variable $maxother$ is used for keeping track of the largest physical clock value received by other processes. The state variable $nextsend$ records when process i is supposed to send its physical clock value to the other processes. The logical clock, $logclock$, is defined to be the maximum of $maxother$ and $physclock$; it is a derived variable (i.e., a variable whose value is derived by state variables).

```
automaton ClockSync (u,r: Real , i: Index )
   signature
      external send (m: Real , const i: Index ),
      receive (m: Real , j: Index , const i: Index)
                where j != i
   states
      nextsend : discrete Real := 0,
      maxother : discrete Real := 0,
      physclock : Real := 0
      initially u > 0 /\ (0 <= r < 1)

   derived variables
   logclock = max (maxother , physclock)
   transitions
      external send (m,i)
      pre m = physclock /\ physclock = nextsend eff
      nextsend := nextsend + u
      external receive (m,j,i)
      eff maxother := max ( maxother ,m)
   trajectories
      stop when physclock = nextsend
      evolve (1 - r) <= d( physclock ) <= (1 + r)
```

Listing 1. Clock Synchronization TIOA

A $send(m, i)$ transition is enabled when $m = physclock$ and $nextsend = physclock$. It causes the value of $nextsend$ to be updated such that the next send occurs when $physclock$ has advanced by u time units. The transition definition for $receive(m, j, i)$ specifies the effect of receiving a message from another process j. Upon the receipt of a message m from j, process i sets $maxother$ to the maximum of m and the current value of $maxother$, thereby updating its knowledge of the largest physical clock value of the other processes. The analog variable $physclock$ does not change at the same rate as real time but it drifts with a rate that is bounded by r. The periodic sending of physical clocks to other processes is enforced through the stopping condition in the trajectory specification; time is not allowed to pass beyond the point where $physclock = nextsend$.

We now present the translation steps in detail.

1. Downloading the toolkit and creating a project. The Tempo toolkit as well as all translated examples including Clock Synchronization can be downloaded from the Tempo website [2]. Installation and configuration information can also be found there; an installation wizard make the process quite straightforward. The toolkit can be installed on Windows 7 OS, Mac-OSX and Linux OS. All examples are located under tab *Samples* as a .zip file, containing all required .tioa to compile a specific algorithm. Once the Tempo toolkit is installed and configured, a project can be created. A link to a video demonstrating a project setup is under tab *Videos*. In our example, we created a project named *ClockSync*.

2. Selecting the communication medium and specifying the algorithm. After a project is created, it can be populated with specification files, which must have a *.tioa* extension. The algorithm to be implemented must be specified in the Tempo language. As mentioned, Tempo is very similar to the TIOA language, which in turn can be learned quite easily (cf. [15]). Before specifying an algorithm in Tempo, the communication medium needs to be decided; depending on the chosen medium, the appropriate automata and vocabularies should be included in the project. Currently, the Tempo translator supports two types of point-to-point channels: (i) a FIFO, reliable channel implemented over MPI, and (ii) a dynamic[1] channel implemented over TCP. (We are working on providing additional channel alternatives, such as multicast channels.) Specification files for both MPI and TCP can be found in [2]. Listing 2 depicts the corresponding Tempo specification of Clock Synchronization Algorithm using MPI as communication protocol between participating processes.

As one may verify by comparing the two versions of the algorithm, the differences between TIOA and Tempo are not significant. For example, u and r parameters that were declared as type of Real in TIOA, are defined in Tempo as DiscreteReal. The Tempo framework provides a set of built-in data types and operations specified as *vocabularies*. However, users can define and import, if needed, abstract data types through the association of types and operations within their own vocabulary definitions. The point-to-point channels are modeled in Tempo as a composition of the `SendMediator` and `ReceiveMediator` automata and based on the chosen communication protocol specific auxiliary automata should be used. In case of MPI, these are the following: *SendMediator.tioa* (Listing 3), *ReceiveMediator.tioa*, (Listing 4), *MPIVocabs.tioa* (Listing 5), and *mpi_message_voc, mpi_request_voc, mpi_status-voc, mpi_voc, alg_specific_voc* as defined in Listing 5.

The communication channels for both protocols are already specified and are part of the translation model. Notice that

[1]The Java API for the TCP socket classes implement convenient methods of sending object data with initially unknown size that changes during execution

```
automaton ClockSync(u, r: DiscreteReal)
  signature
    output SEND( m:Null[mpi_message] )
    input  RECEIVE( m:Null[mpi_message] )
    internal prepmessages, init
  states
    nextsend : DiscreteReal := 0;
    maxother : DiscreteReal := 0;
    physclock : DiscreteReal := 0;
    tosend : Seq[Null[mpi_message]] := { };
    rates : Array[Nat,DiscreteReal] := constant(1);
    index : Nat := 0;
    initially u >= 0 /\ (0 <= r /\ r <= 1);
  transitions
    output SEND(m) where len(tosend) ~= 0
      pre m = head(tosend);
      eff tosend := tail(tosend);
    input RECEIVE(m)
      eff  if (m ~= nil()) then
           maxother := max(maxother, val(m).data); fi
    internal prepmessages
      pre len(tosend) = 0;
      eff nextsend := nextsend + u;
          for n : Nat where n < MPI_Size() do
            tosend := tosend |- embed([physclock, n]);
          od
          index := mod(index + 1, 50);
    internal init
      locals v : DiscreteReal := 0; b : Bool := false;
      pre true;
      eff
         for n : Nat where n < 50 do
            v := choose n; % where 1-r <= n /\ n < 1 + r;
            if v > r then v := r; fi
            b := choose n;
            if b then
                rates[n] := rates[n] + 1 + v;
            else
                rates[n] := rates[n] + 1 - v;
                fi
         od;

  trajectories
    trajdef T stop when physclock > nextsend;
    evolve d(physclock) = rates[index];
```

Listing 2. Clock Synchronization Tempo version

```
automaton SendMediator
    signature
        input SEND(m:Null[mpi_message])
    states
        status:Array[Nat, Null[mpi_request]] :=
        constant(nil());
        clock:AugmentedReal := 0;
    transitions
        input SEND(m)
        eff
        if (m ~= nil()) then
            status[val(m).destination] := MPI_Isend(val(m),
            val(m).destination);
        fi
  trajectories
    trajdef DELAY evolve d(clock) = 1;
```

Listing 3. SendMediator TIOA

the type of a parameter in the send and receive actions is `Null[mpi_message]`. This means that the legal values of these parameters can be all values in `mpi_message` plus the special value `nil`. When passing a `nil` value to the send action, this results in a no-operation semantics as the send mediator will not send a `nil`-message. The receive mediator on the other hand will return a `nil` message in the case where there are no more messages pending delivery.

Guidelines and more information about the Tempo language and the mediator automata can be found in [14].

3. Composition. In the Tempo model, all auxiliary mediator automata created to implement abstract system services are

```
automaton ReceiveMediator
signature
    output RECEIVE(m:Null[mpi_message])
    input  probe(s:Nat)
states
    toRecv:Seq[mpi_message] := {};
transitions
    output RECEIVE(m) where len(toRecv) = 0
    pre m = nil();
    output RECEIVE(m) where len(toRecv) ~= 0
    pre m = embed(head(toRecv));
    eff toRecv := tail(toRecv);

    input probe(s)
    locals status:Null[mpi_status] := nil();
    eff
        status := MPI_Iprobe(s);
        if (status ~= nil()) then
           if (MPI_Test(val(status))) then
              toRecv := toRecv |- MPI_Irecv(val(status),s);
           fi
        fi
```

Listing 4. ReceiveMediator TIOA

```
%%% .: algorithm vocabs :.
vocabulary alg_specific_voc
        types %Index : Enumeration [p1, p2, p3, p4],
        Data  : DiscreteReal
end

vocabulary mpi_voc
   operators MPI_Rank : -> Nat, MPI_Size : -> Nat
end

vocabulary mpi_request_voc
   imports mpi_message_voc
   types mpi_request
   operators
   MPI_Isend : mpi_message, Nat -> Null[mpi_request],
   MPI_Barrier : -> Bool
end

%%% .:MPI Channel vocabs:.
vocabulary mpi_status_voc
        types mpi_status
        operators MPI_Iprobe :
                Nat -> Null[mpi_status],
                MPI_Test : mpi_status -> Bool
end

%%% .: algorithm vocabs :.
vocabulary mpi_message_voc
        imports alg_specific_voc, mpi_status_voc
        types mpi_message : Tuple[data:Data,destination:Nat]
        operators
        MPI_Irecv : mpi_status, Nat -> mpi_message
end
```

Listing 5. MPI Vocabularies

combined with the source automaton through *composition*; this is the next step in the process of translating an algorithm from Tempo to Java. The composition happens at runtime, where all automata interfaces and pass parameters are matched in the process. Specifically, our compiler performs action matching of the composed automata within its components. Action matching is restricted to actions that have fixed and a priori known parameters with strict one-to-one type correlation. The user only needs to create a new .tioa file, importing and including all required automata for composition. In our example, *ClockSync.tioa* should be composed with *MPIVocabs.tioa, ReceiveMediator.tioa* and *SendMediator.tioa* resulting to the new automaton *ClochSyncSys.tioa* as shown in Listing 6 which will be fed into to the tempo to java compiler (after its syntax is checked). Prior to this, non-determinism should be resolved as we explain next.

```
%%% .: VOCABS :.
include "MPIVocabs.tioa"
imports mpi_message_voc, mpi_request_voc, mpi_status_voc,
        mpi_voc, alg_specific_voc

%%% .: ALG AUTOMATON :.
include "ClockSync.tioa"

%%% .: MPI AUTOMATA :.
include "ReceiveMediator.tioa", "SendMediator.tioa"

automaton ClockSyncSys(u, r: DiscreteReal)
  components
    CS : ClockSync(u,r); SM : SendMediator;
    RM : ReceiveMediator;

schedule
   states m : Null[mpi_message] := nil(); runs : Nat := 100;

do
   fire internal CS.init;
   for i :Nat where i < runs do
      follow CS.T duration \infty;
      fire internal CS.prepmessages;
      for j :Nat where j < MPI_Size() do
         fire output CS.SEND(m);
      od

      follow SM.DELAY duration (MPI_Size() * 10);

      for j :Nat where j < MPI_Size() do
         m := nil();
         fire input RM.probe( j );
             fire output RM.RECEIVE( m );
od od od
```

Listing 6. Clock Synchronization Composed Tempo version

4. Schedule. Since the Tempo framework is inherently nondeterministic, translating programs written in Tempo language into an executable language such as Java requires resolving all nondeterministic behaviors. This process is called *scheduling* and is the last but not least step before the translation. Developing a method to schedule automata, which is a metadata appended to the automaton that we desire to execute, is the key conceptual challenge in the initial design of our code generator. In general, it is computationally infeasible to schedule Tempo programs automatically. Instead, Tempo provides constructs specified in the nondeterminism resolution language (NDR) [25] that enable resolution of nondeterminism through which developers can schedule automata directly and safely. This schedule outlines the sequence of actions that will be invoked by nodes of the system during execution. Usually, it is sufficient to schedule actions in a round-robin fashion (which yields fair schedules); learning to write such schedules requires little effort. The schedule we developed for the Clock Synchronization algorithm is presented in Listing 7**.** A node, i, participating in the clock synchronization system will behave according to its schedule.

5. Checking the syntax. The front-end of the tool is used to parse the final specification and check its syntactic correctness. If there are no errors the checked specification can be proved to be correct, either manually or using the automated theorem prover provided by the toolkit (as mentioned, this is beyond the scope of this work). If the source specification will be used to generate networked code (as with our example), then prior to generating Java code the user must configure the translation module with the type of the channel to be used (either MPI or TCP) (of course this selection must match the one in Step 2).

6. Compilation. The translation process from Tempo to Java is performed either by selecting the corresponding button in the

```
schedule
    states
    m : Null[mpi_message] := nil();
    runs : Nat := 100;
do
    fire internal CS.init;
    for i :Nat where i < runs do
        follow CS.T duration \infty;
            fire internal CS.prepmessages;

        for j :Nat where j < MPI_Size() do
            fire output CS.SEND(m);
            od

        follow SM.DELAY duration (MPI_Size() * 10);

        for j :Nat where j < MPI_Size() do
            m := nil();
            fire input RM.probe( j );
            fire output RM.RECEIVE( m );
        od
    od
od
```

Listing 7. Clock Synchronization Schedule

GUI or through *Tools* $\rightarrow$ *Run tempo2java*. After compilation in the navigator pane, a new directory will appear which is the folder that includes the generated Java code. The name of this directory is derived from the top level automaton found in the project. In our case, the main automaton is called `ClockSyncSys.tioa`, and hence the generated directory is named `ClockSyncSys.java`. Eclipse Java development environment can be used to verify and compile the translated Java code (from the `ClockSyncSys.java` directory). Note that in addition to the GUI, the Tempo toolkit also provides a command line interface, where tools are selected using a parameter flag along with any other tool specific parameters, for example, `-plugin=tempo2java -comm=MPI`.

Restrictions. The expressiveness of Java and hardware limitations impose restrictions on the set of Tempo (source) specifications admitted for translation. For instance, the source specifications must be in the node-channel form [12] where the specification provides components running on a node, and it interfaces with other nodes via a network protocol, such as MPI or TCP [14] (all specifications presented in this paper follow this form). Moreover, it is the responsibility of the Tempo programmer to decide on the distribution of computation for the intended deployment setting. Given the generality of the Tempo framework, each networked component can run a different node algorithm, hence in that case compilation proceeds on a node-by-node basis. Alternatively, a node algorithm could be general enough to allow it to be executed on any number of nodes. Java, as the target programming language, ensures portability of the generated programs (through specialized Java Virtual Machines) and hides architectural specifics.

We now discuss some restrictions related to handling time. In TIOA, all actions take zero time (they are instantaneous) which is not true for the actual implementation. The duration of discrete transitions is not accounted for in the trajectory. However, this is not a real problem since trajectories, are already discussed, are with respect to logical time, not physical time. As an example, consider the receive action, where if there are no messages to be received, then the receiver will time-out while waiting on the socket. Another timing issue to consider is that due to the discrete nature of Java types, the trajectory bounds must be mindful of floating point precision.

The translation process of Tempo to Java along with low-level technical issues are presented in depth in [14]. Next we argue that the generated Java code can be viewed as a specialization of the source specification, and that the set of its allowed behaviors solves the same problems as the source specification. Hence, the resulted code is proved *correct by construction*.

IV. Translation's Correctness

We now overview the translation's correctness. The complete proof is given in [14]. The proof of correctness follows [27], where it presents a framework for an incremental proof technique for untimed systems that was applied, among other things, to a translation of a *group communication service* into its implementation using C++.

Throughout this section we refer to the source model as a *parent specification* (or simply *parent*) and the abstracted Java code derived by translating the *parent* specification as a *child implementation* (or simply *child*). The work in [27] introduced the *specialization* and *extension* operations on the parent which result in the child implementation; it can be shown that following application of these operations on the parent the derived child can be used anywhere the parent can be used. *Here we extend these operations to support timed systems* (this is one of our key contributions). We begin with a formal definition of TIOA as presented in [3].

Definition 1: A *timed automaton* A is defined as $A = (X, \mathcal{Q}, \Theta, E, H, \mathcal{D}, \mathcal{T})$, s.t. X is a set of *internal variables*; $\mathcal{Q} \subseteq val(X)$ is a set of *states*, where informally $val(X)$ represents the set of values over types of X; $\Theta \subseteq \mathcal{Q}$ is a nonempty set of *start states*; E is a set of *external actions* and H is a set of *internal* actions, where $E \cap H = \emptyset$; $\mathcal{D} \subseteq \mathcal{Q} \times (E \cup H) \times \mathcal{Q}$ is a set of *discrete transitions*; and $\mathcal{T} \subseteq trajs(Q)$ is a set of trajectories, where informally $trajs(Q)$ represents the set of all trajectories for variables in X.

Trajectory axioms: For some $\tau \in \mathcal{T}$ we associate states, such that $\tau.fstate = x \in \mathcal{Q}$ at the start of τ. If τ is closed, then $\tau.lstate = x' \in \mathcal{Q}$ where x' is the state of A at the end of τ and $\tau.ltime$ is the duration of τ. When $\tau.fstate = x$ and $\tau.lstate = x'$ and $x, x' \in \mathcal{Q}$, then the following axioms must hold: **T0** If $x \in \mathcal{Q}$ then there exists a *point trajectory* τ such that duration of τ is zero and $\tau.fstate = \tau.lstate = x$. **T1** For every $\tau \in \mathcal{T}$ and every $\tau' \leq \tau$, $\tau' \in \mathcal{T}$, asserting a prefix closure. **T2** For every $\tau \in \mathcal{T}$ and every t in the domain of τ, $\tau \trianglerighteq t \in \mathcal{T}$, asserting a suffix closure (i.e., that the remainder is also a trajectory). **T3** Let $\tau_0\ \tau_1\ \tau_2 \ldots$ be a sequence of trajectories in $\mathcal{T}$ such that, for each nonfinal index i, τ_i is closed and $\tau_i.lstate = \tau_{i+1}.fstate$. Then $\tau_0 \frown \tau_1 \frown \tau_2 \ldots \in \mathcal{T}$, asserting a concatenation closure.

A *forward simulation* relates states of two automata (from [3]), which asserts that each discrete transition (resp. trajectory) of A can be simulated by a corresponding execution fragment of S with the same trace and duration, and leads to the trace inclusion property.

Definition 2: Let A (child) and S (parent) be two automata with the same external signature. A relation $\mathcal{R} \subseteq Q_A \times Q_S$ is a *forward simulation* from A to S if it satisfies the following three conditions: `(1)` If $t \in \Theta_A$, then there exists state $s \in \Theta_S$ such that $(t, s) \in \mathcal{R}$. `(2)` If $(t, s) \in \mathcal{R}$ and α is an execution fragment of A consisting of an action surrounded by two point trajectories, with $\alpha.fstate = t$, then S has a closed execution fragment β with $\beta.fstate = s$, $\texttt{traces}(\beta) = \texttt{traces}(\alpha)$, and $(\alpha.lstate, \beta.lstate) \in \mathcal{R}$. `(3)` If $(t, s) \in \mathcal{R}$ and α is an execution fragment of A consisting of a single closed trajectory, with $\alpha.fstate = t$, then S has a closed execution fragment β with $\beta.fstate = s$, $\texttt{traces}(\beta) = \texttt{traces}(\alpha)$, and $(\alpha.lstate, \beta.lstate) \in \mathcal{R}$.

The *specialization operation* presented in [27] is a construct for creating a child by specializing the parent. This operation is designed to capture the notion of subtyping in I/O automata in the sense of trace inclusion. The child can read parent's state, add new (read/write) state components, and restrict parent's transitions. The *specialize* construct operates on the parent and the following additional parameters: a *state extension*, the new state components, an *initial state extension*, the initial values of the new state components, and a *transition restriction*, specifying the child's addition of new preconditions and effects (modifying new state components only) to parent transitions. We extend the *specialize* construct with a *trajectory restriction*, specifying the child's restrictions on stopping conditions of parent's trajectories. Formally stated:

Definition 3: (Specialization) Let $\texttt{A} = (X, \mathcal{Q}, \Theta, E, H, \mathcal{D}, \mathcal{T})$ be an automaton; X_n a set of variables and $N \subseteq val(X_n)$ be any set of states, called a *state extension*; N_0 be a non-empty subset of N, called an *initial state extension*; and, $TranR \subseteq (\texttt{states(A)} \times N) \times \texttt{sig(A)} \times N$ be a relation called a *transition restriction*. Let V_X be a set of variables in $\texttt{states(A)}$, V_N be a set of variables in N, $V \in V_X$ be such that for each variable $v \in V$ its type is dynamic (i.e., $dtype(v)$), then $TrajR \subseteq \{\tau : dom(\tau) \rightarrow dtype(v) \wedge dom(\tau) \rightarrow type(v), \forall v \in V \wedge \forall v' \in V_N\}$ be a *trajectory restriction*[1]. Then $\texttt{specialize(A)}(X_n, N, N_0, TranR, TrajR)$ defines $\texttt{A}' = (X', \mathcal{Q}', \Theta', E', H', \mathcal{D}', \mathcal{T}')$:

- $X' = X \cup X_n$, $\mathcal{Q}' = \mathcal{Q} \times N$, $\Theta' = \Theta \times N_0$, $E' = E$ and $H' = H$
- $\mathcal{D}' = \{(\langle t_p, t_n \rangle, \pi, \langle t'_p, t'_n \rangle) : (t_p, \pi, t_{p'}) \in \mathcal{D} \wedge (\langle t_p, t_n \rangle, \pi, t'_n) \in TranR\}$, where $\langle t_p, t_n \rangle$ is a state in $\mathcal{Q}'$,
- $\mathcal{T}' \subseteq TrajR$ the set of all trajectories for variables in X'.

For an automaton `A'` as defined above, given $t \in \mathcal{Q}'$, we write $t|_p$ to denote its parent component and $t|_n$ to denote its new component. If α is an execution fragment of `A'`, then $\alpha|_p$ and $\alpha|_n$ denote sequences obtained by replacing each state t in α with $t|_p$ and $t|_n$, respectively.

We now reintroduce the *specialized extension* operation from [27]. This operation is similar to inheritance, where the child cannot overwrite parent's behaviors, but can extend these with new types of behaviors. Specialized extension is performed in two steps, first via *signature extension* and then by the application of the previously presented *specialization* operation. The signature extension operation extends parent with new actions, where these are enabled in every state and do not modify the state, where specialization gives meaning to these actions. The result is a child automaton with extended signature, but same states and same start states. Additionally, state extension operation allows action renaming, which is specified by a *signature-mapping* function that maps child's actions to parent's actions. This mapping can be many-to-one,

[1]For a formal definition of dynamic, $dtype$, and static, $type$, see [3].

onto, and is undefined for new actions added. For example, let f be a *signature-mapping* then for each action in parent's signature there is at least one corresponding $f(.)$, and if π is a new action under signature extension then $f(\pi) = \perp$.

Definition 4: Assume $\mathtt{A} = (X, \mathcal{Q}, \Theta, E, H, \mathcal{D}, \mathcal{T})$, $\mathcal{S}'$ to be some signature. Let f be a partial function, called *signature-mapping*, from $\mathcal{S}'$ to signature of $\mathtt{A}$ such that f is onto and preserves the classification of actions. Then, $\mathtt{extend(A)}(\mathcal{S}', f)$ is defined to be the following automaton $\mathtt{A}' = (X', \mathcal{Q}', \Theta', E', H', \mathcal{D}', \mathcal{T}')$:

- $X' = X$, $\mathcal{Q}' = \mathcal{Q}$, $\Theta' = \Theta$, $\{E' \cup H'\} = \mathcal{S}'$, $\mathcal{T}' = \mathcal{T}$
- $\mathcal{D}' = \{(t, \pi, t') \in \mathcal{Q}' \times \mathcal{S}' \times \mathcal{Q}' : ((f(\pi) = \perp) \wedge (t = t')) \vee ((f(\pi) \neq \perp) \wedge ((t, f(\pi), t') \in \mathcal{D}))\}$

Hence, $\mathtt{A}'$ is a *signature extension* of $\mathtt{A}$ with signature-mapping f if $\mathtt{A}'$ is such that $\mathtt{A}' = \mathtt{extend(A)(sig(A')}, f)$ for some signature-mapping f from signature of $\mathtt{A}'$.

Definition 5: Automaton $\mathtt{A}'$ is a *specialized extension* of $\mathtt{A}$ if $\mathtt{A}'$ is a specialization of a signature extension of $\mathtt{A}$.

We are now ready to restate the final result from [27] in terms of timed automata. In the theorem that follows we assume that each automaton Aut is defined as $\mathtt{Aut} = (X_{\mathtt{Aut}}, \mathcal{Q}_{\mathtt{Aut}}, \Theta_{\mathtt{Aut}}, E_{\mathtt{Aut}}, H_{\mathtt{Aut}}, \mathcal{D}_{\mathtt{Aut}}, \mathcal{T}_{\mathtt{Aut}})$.

Theorem 1: Let $\mathtt{A}'$ be a specialized extension of $\mathtt{Aut}$ with a signature-mapping f. Let $\mathtt{S}'$ be a specialized extension of $\mathtt{S}$ with a signature-mapping g, such that $\mathtt{S}' = \mathtt{specialize(extend(S(G}, g)))(N, N_0, TR)$. Assume that $\mathtt{Aut}$ and $\mathtt{S}$ have the same external signatures and that $\mathtt{Aut}$ implements $\mathtt{S}$ via a simulation relation R_p. Assume further that $\mathtt{A}'$ and $\mathtt{S}'$ have the same external signatures, and that, for every external action $\pi \in \mathtt{A}'$, $g(\pi) = f(\pi)$.

A relation $\mathcal{R}_c \subseteq \mathcal{Q}_{\mathtt{A}'} \times \mathcal{Q}_{\mathtt{S}'}$, defined in terms of relation $\mathcal{R}_p$ and a new relation $\mathcal{R}_n \subseteq \mathcal{Q}_{\mathtt{A}'} \times N$ as $\{(t, s) : (t|_p, s|_p) \in \mathcal{R}_p \wedge (t, s|_n) \in \mathcal{R}_n\}$, is a simulation from $\mathtt{A}'$ to $\mathtt{S}'$ if $\mathcal{R}_c$ satisfies the following two conditions:

1 For every $t \in start(\mathtt{A}')$, there exists a state $s|_n \in \mathcal{R}_n(t)$ such that $s|_n \in N_0$.

2 If t is a reachable state of $\mathtt{A}'$, s is a reachable state of $\mathtt{S}'$ s.t. $s|_p \in \mathcal{R}_p(t|_p)$ and $s|_n \in \mathcal{R}_n(t)$, and α is an execution fragment of $\mathtt{A}'$ where $\alpha.lstate = t'$ and consisting of one action surrounded by two point trajectories, τ_0 and τ_0' ($\tau_0.lstate = s_i$ and $\tau_0'.fstate = s_{i+1}$), or a single closed trajectory τ, then there exists a closed execution fragment β of $\mathtt{S}'$ beginning from s and ending at some state s', and satisfying the following: (a) $\beta|_p$ is an execution fragment of S. (b) For every step (s_i, σ, s_{i+1}) in β, $(s_i, \sigma, s_{i+1}|_n) \in TR$. (c) $s'|_p \in R_p(t'|_p)$. (d) $s'|_n \in R_n(t')$. (e) β has the same trace as $\tau \xrightarrow{\pi} \tau'$. (f) If $\alpha = \tau$, then for each $\tau_i \in \beta$, $\sum_i \tau_i.ltime = \tau.ltime$.

The above constitutes the core result that we use to prove our translator's correctness. *We emphasize that this result is general and it can be used beyond our work.*

Translation soundness. Since we cannot use Java's API in proofs, we have to bring the translator generated codes into the Tempo framework by providing an abstraction of the executable code. In [12], techniques are presented that demonstrate how to abstract the generated Java code by the IOA translator and prove that the resulting code has the same externally observable behaviors as the source specification, where this is done by first reasoning about translation of the individual node automata (see Theorems 7.1 [12]), and then at the composition level, i.e. system wide, (see Theorems 7.2 in [12]). Same techniques can be adapted to this work and used to arrive at the same conclusion for the codes generated by our Tempo to Java translator. To avoid tedious repetition we forgo presentation of this detail and refer the interested reader to [12] and [14].

In the context of translating timed automata models into executable code, the input model is $\mathtt{S}$ and its implementation is $\mathtt{A}'$. Given the full details, it is easy to see that our translation performs *specialization* and *signature extension* on the source specification. It remains to show that these are consistent with the proceeding results.

Theorem 2: The Tempo to Java translator preserves behaviors of the source specification.

Proof sketch: Let $\mathtt{A}$ be an abstracted automaton from the Java code generated by the direct translation of the allowed Tempo syntax into the Java code. Let $\mathtt{A}' = \mathtt{specialize(extend(A(G}, f)))(N, N_0, TR)$. Given $\mathtt{A}$ and $\mathtt{A}'$, the claim follows from Thm. 7.2 of [12] and Thm. 1.

It may be surprising to the reader that a proof for an untimed model can be so directly applied to the translation of a timed system. Thus some additional elaboration is needed. Once we obtain the abstracted Java automaton from Theorem 1 we are left with a Java code that represents the source automaton with the added restrictions of the Java programming language and JVM. The notable new addition is that of trajectories, however at this point coded trajectories are a special kind of transitions where are periodically sampled over some interval until stopping condition or the end of the interval is reached. Therefore the remaining step is to actually reason about its execution at the JVM level. Theorem 7.2 [12] exactly addresses that step and provides the machinery to be used for reasoning about the individual Java instructions cannot violate the established relation.

Finally, recall that we are in a node-channel model and execution of all trajectories is per node. Moreover, the time is only logical. Therefore, since the same clock is used to drive the sampling of all trajectories (in the on-node-composition) the rate at which the logical time is "synchronized" to the rate of the local physical clock for the JVM of the host node. Hence the logical timing behavior is preserved. ■

V. Implementation

Here we present some experimental results we have obtained after applying the methodology we have described. This is just a simple proof-of-concept implementation that demonstrates that our methodology works and produces implementations with reasonable performance; it is not by any means an evaluation study of the algorithm. We have used a version of clock synchronization system specialized to MPI channels and and one specialized to TCP channels. The target platform consists of a cluster of nine machines, hosted at the University of Cyprus. Each machine is powered by an AMD Opteron 2.5GHz (single or dual) CPU and is running Linux (CentOS v5.5).

Given the specification from Section III where all source and vocabulary files are placed in some directory the Java code can be obtained using a GUI or a command prompt and by running: `tempo.sh -plugin=tempo2java -comm=MPI myfile.tioa`. For the TCP-based model the -comm flag is TCP.

TABLE I. MPI AND TCP IMPLEMENTATION: VALUE OF MAX($physclock$,$otherclock$) AT TERMINATION.

MPI	Time (ms)	TCP	Time (ms)
node 1	59401.32	node 1	59400.37
node 2	59401.32	node 2	59400.29
node 3	59401.32	node 3	59400.31
node 4	59401.80	node 4	59401.08
node 5	59400.26	node 5	59400.85
node 6	59401.17	node 6	59400.85
node 7	59401.17	node 7	59400.85
node 8	59400.58	node 8	59400.29
node 9	59402.29	node 9	59400.85

Recall the specification of the clock synchronization system from List. 2. In our experiments we choose $u = 600$ (milliseconds) and $r = 0.6$. Each node repeats the algorithm's steps 100 times and the final clock values are recorded. To execute the MPI version of the code one would use the following command `mpjrun.sh -np 9 -dev niodev -ea ClockSychSys 600 0.6`, where we assume that MPJ [26] has been installed correctly.

Results in Table I represent the final values of $physclock$ at each system participant and for both version of the implementation. The first observation is that the values for both systems are roughly similar, which is exactly what we expect. The second observation is that for each system, the clocks of individual nodes are approximately equal, where the difference between the maximum and minimum is 2.03 for MPI and 0.79 for TCP. This discrepancy can be explained by the fact that the translation does not make any attempt at synchronization across the nodes, since the source specification does not require it. Per step 1. of the algorithm and natural differences in the speed of the physical machines, some nodes inevitably execute faster than others, terminate sooner, and stop sending messages, therefore, making it impossible for the slower nodes to synchronize with the fast ones and vice versa.

VI. CONCLUSIONS

There are two practical challenges when modeling with Tempo. First is resolving nondeterminism and providing nodes' schedules. Specifically, the node schedule should be such that progress is guaranteed despite lack of synchronization across nodes and while maintaining the timing guarantees of the specification. The channel automata have been designed with as uniform interface as possible and the key differences appear in the schedule. There may be an initial conceptual difficulty with Tempo due to its embedded nondeterminism and ability to describe systems with multiple correct behaviors. This may be a departure from the deterministic control-flow diagrams used during software design. An attribute of Tempo is that its type system and structure of control statements are similar to the modern imperative programming languages. Finally, the composition operation on automata can be viewed as a case of hierarchical inheritance which is a familiar concepts in object-oriented programming.

As already mentioned, the translation module has been tested both at MIT and the University of Cyprus at undergraduate and graduate level. After some initial difficulty, there was no particular problem, neither with the compositional philosophy of Tempo, nor with the need for writing schedules. We have received similar comments from students and researchers from other institutions that have used the translation tool.

The Tempo toolkit is being actively updated, and we are continuing to improve and extend current functionality. One of our main aims is to make the generated code as efficient as possible and improve its human readability. To this respect, we plan to measure the readability of the generated code by designing and deploying questionnaire assignments in the distributed computing courses of our (and possible other) institutions.

REFERENCES

[1] N. Lynch, L. Michel, and A. A. Shvartsman. Tempo: A toolkit for the timed input/output automata formalism. *Proc. of SIMUTools 2008*.

[2] VeroModo, Inc. http://www.veromodo.com.

[3] D. K. Kaynar, N. Lynch, R. Segala, and F. Vaandrager, *The Theory of Timed I/O Automata*, 2nd edition, Synthesis Lectures on Distributed Computing Theory. Morgan & Claypool Publishers, 2010.

[4] N. Lynch, *Distributed Algorithms*. Morgan Kaufmann Publishers, 1996.

[5] S. Gilbert, N. A. Lynch and A. A. Shvartsman. RAMBO: A robust, reconfigurable atomic memory service for dynamic networks. *Distributed Computing,* 23(4):225–272, 2010.

[6] R. De Prisco. *Revisiting the Paxos algorithm.* MS thesis, MIT, 1991.

[7] H. E. Bal, M. F. Kaashoek, and A. S. Tanenbaum. Orca: A language for parallel programming of distributed systems. *IEEE Transactions on Software Engineering*, 1992.

[8] M. Hayden. Ensemble reference manual. Cornell University, 1996.

[9] N. A. Lynch, S. J. Garland, D. Kaynar, L. Michel, and A. Shvartsman. *The Tempo Language User Guide and Reference Manual,* http://www.veromodo.com/resources/Tempo_Guide.pdf, 2011.

[10] K. G. Larsen, P. Pettersson, and W. Yi. Uppaal in a nutshell. *Int. Journal on Software Tools for Technology Transfer*, 1:134–152, 1997.

[11] S. Owre, S. Rajan, J. Rushby, N. Shankar, and M. Srivas. PVS: Combining specification, proof checking, and model checking. *Proc. of CAV 1996*, pp. 411–414.

[12] J. Tauber. *Verifiable compilation of I/O Automata without global synchronization*. PhD thesis, MIT, 2005.

[13] C. Georgiou, N. Lynch, P. Mavrommatis, J. Tauber. Automated implementation of complex distributed algorithms specified in IOA. *Intl J. on Soft. Tools for Techn. Transfer*, 11(2):153–171, 2009.

[14] P. Musial. *Using Timed Input/Output Automata for Implementing Distributed Systems.* Technical report, Cambridge, MA, USA, 2011. http://www.veromodo.com/resources/Tempo2JavaGuide.pdf

[15] M. Gelastou, Ch. Georgiou, and A. Philippou. On the application of formal methods for specifying and verifying distributed protocols. *Proc. of NCA 2008*, pp. 195-204.

[16] IOA language and toolset. http://theory.lcs.mit.edu/tds/ioa/.

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

[18] F. W. Vaandrager. On the relationship between process algebra and Input/Output automata. *Proc. of LICS 1991*, pp. 387–398.

[19] R. Cleaveland, J. Gada, P. Lewis, S. A. Smolka, O. Sokolsky, and S. Zhang. The concurrency factory: Practical tools for specification, simulation, verification, and implementation of concurrent systems. *Proc. of DIMACS Workshop on SPA*, pp. 75–89, 1994.

[20] S. Allen, M. Bickford, R. Constable, R. L. Eaton, C. Kreitz, L. Lorigo, and E. Moran. Innovations in computational type theory using Nuprl. *Journal of Applied Logic*, 4(4):428–469, 2006.

[21] Nuprl home page. http://www.nuprl.org.

[22] J. Hickey, N. A. Lynch, and R. van Renesse. Specifications and proofs for ensemble layers. *Proc. of TACAS 1999*, pages 119–133.

[23] A Tool for Modeling and Implementation of Embedded Systems, http://www.timestool.com.

[24] R. Alur and D.L. Dill. A theory of timed-automata. *Theoretical Computer Science,* 126(2):183–235, 1994.

[25] J. Ramrez-Robredo. *Paired Simulation of I/O automata*. MS thesis, MIT, 2000.

[26] M. Baker, B. Carpenter, and A. Shafi. MPJ Express: Towards Thread Safe Java HPC. In Proc. of Cluster Computing, pages 25–28, 2006.

[27] D. Keidar, R. Khazan, N. Lynch, and A. Shvartsman. An inheritance-based technique for building simulation proofs incrementally. *ACM Trans. on Soft. Engin. and Method.*, 11(1):63–91, 2002.